VŠB – Technical University of Ostrava

Faculty of Electrical Engineering and Computer Science

Department of Cybernetics and Biomedical Engineering

Asynchronous Database Access Library with Silent Automatic Reconnections

Asynchronní knihovna pro přístup k databázi s tichým automatickým opětovným propojením

2019 Štěpán Němec

VŠB - Technical University of Ostrava
Faculty of Electrical Engineering and Computer Science
Department of Cybernetics and Biomedical Engineering

Diploma Thesis Assignment

Student: **Bc. Štěpán Němec**

Study Programme: N2649 Electrical Engineering

Study Branch: 2612T041 Control and Information Systems

Title: Asynchronous Database Access Library with Silent Automatic Reconnections
Asynchronní knihovna pro přístup k databázi s tichým automatickým opětovným propojením

The thesis language: English

Description:

The aim of the diploma thesis is to design and implement the library for asynchronous access to the database. In this particular case, it is the implementation of parallel queries from specific HMI applications and the Firebird relational database.

In summary, the thesis is characterized by the following points:
1. Analysis of the current state from the point of view of the current access to database transitions.
2. Analysis, design and implementation of asynchronous polling.
3. Analysis, design and implementation of the demonstration application.
4. Testing of essential functional units and demonstration application.
5. Evaluation of achieved results.

References:

[1] CLARKE, Gordon a Deon REYNDERS. *Practical Modern SCADA Protocols: DNP3, 60870.5 and Related Systems.* Oxford: Newnes, 2004. ISBN 978-0750657990.
[2] CORONEL, Carlos a Steven MORRIS. *Database Systems: Design, Implementation, & Management.* Boston: Cengage Learning, 2018. ISBN 978-1337627900.
[3] GABRIJELCIC, Primoz. *Delphi High Performance: Build fast Delphi applications using concurrency, parallel programming and memory management.* Birmingham: Packt Publishing, 2018. ISBN 978-1788625456.
[4] SIPPU, Seppo a Eljas SOISALON-SOININEN. *Transaction Processing: Management of the Logical Database and its Underlying Physical Structure (Data-Centric Systems and Applications).* Cham: Springer, 2015. ISBN 978-3319122915.
[5] LABOON, Bill. *A Friendly Introduction to Software Testing.* CreateSpace Independent Publishing Platform, 2016. ISBN 978-1523477371.

Extent and terms of a thesis are specified in directions for its elaboration that are opened to the public on the web sites of the faculty.

Supervisor: **Ing. Zdeněk Slanina, Ph.D.**

Date of issue: 01.09.2018

Date of submission: 30.04.2019

doc. Ing. Jiří Koziorek, Ph.D.
Head of Department

prof. Ing. Pavel Brandštetter, CSc.
Dean

I hereby declare that this master's thesis was written by myself. I have quoted all the references I have drawn upon.

30 April 2019 .....

I want to thank my supervisor **Ing. Zdeněk Slanina, Ph.D.** for his conscientious guidance and excellent support during the thesis. I also would like to thank colleagues from **Elekt Labs s.r.o.** for their helpful advice and practical experiences that helped me find the way to the final solution.

I hereby agree to the publishing of the master's thesis as per s. 26, ss. 9 of the Study and Examination Regulations for Master's Degree Programmes at VŠB-Technical University of Ostrava.

24 April 2019

Abstrakt

V dnešní době je v produkčních aplikacích použit synchronní přístup k databázi. V synchronním přístupu je hlavní vlákno blokováno, dokud není SQL dotaz vykonán, což v případě dlouho běžících SQL dotazů, či problémů na síti způsobuje zasekávání aplikací. Diplomová práce se pro nalezení řešení tohoto problému zabývá možnostmi a principy výměny dat mezi Firebird databází a aplikacemi vyvinutými v programovacím jazyce Delphi. Cílem této diplomové práce je databázová knihovna pro asynchronní přístup k databázi, kde je možné vykonávat několik dotazů paralelně. Databázová knihovna také obsahuje funkce pro automatické tiché připojení, automatické testování a další, aby byla co nejvíce komfortní pro programátory, kteří ji budou používat.

Klíčová slova: Firebird, Delphi, asynchronní databázový přístup, vícevláknové zpracování

Abstract

Nowadays in production applications is used synchronous access to the database. In synchronous access is main thread blocked until SQL statement is not executed which in case of long-running SQL query or network problem causes application jamming. The diploma thesis deals with possibilities and the principles of data exchange between the Firebird database and applications developed in Delphi programming language to find a solution for this problem. The aim of this thesis is a library for asynchronous database access, where to perform multiple requests in parallel form is possible. The database library also contains functions for automatic silent reconnections, automatic testing and others to make it as comfortable as possible for programmers who will use it.

Keywords: Firebird, Delphi, asynchronous database access, multithread processing

Table of contents

List of symbols and abbreviations used

ACID	atomicity, consistency, isolation, durability
API	application programming interface
CPU	central processing unit
DSQL	dynamic structured query language
GUI	graphical user interface
HMI	human machine interface
I/O	input / output
ID	identifier
NetBEUI	network basic input out system extended user interface
PLC	programmable logic controller
SQL	structured query language
SQL92	third revision of structured query language
TCP/IP	transmission control protocol / internet protocol
XML	extensible markup language

List of figures

List of tables

1. Preface

Currently in all applications, which are responsible either for gathering data from PLCs, or production management in factories, is used synchronous access to the database. At the beginning of application development, this access to the database was enough to all requirements of application functionality. However, how customers increase requirements for application functionality, in the same way is increased the burden for applications. Now, in some situations, is this burden so significant, that some more extended performance of SQL statement can cause freezing of HMI panel because main thread still works on SQL statement and cannot perform other actions. Situations like this are very unpleasantly for operators, and with using asynchronous access to the database, they can disappear entirely. Description of asynchronous database access and others I/O operations is in chapter 2.

In chapter 3 is described how work Firebird database to which is linked next chapter about transactions. In this chapter is described what transactions are, why to use them and some transaction models, which should be used for the solution of a diploma thesis because in the final database library is required to execute several SQL statements in parallel threads. Next chapter is devoted to Delphi programming language in which is diploma thesis written. This chapter focuses mainly on programming techniques, which should be used in the solution of the diploma thesis. Chapter 6 is the last chapter devoted to theory, and there are described techniques for software testing.

From chapter 7 begins the description of the solution of the diploma thesis. Firstly is described topology of the solution where are outlined main components of the database library. In the next two chapters are described two main parts of the database library, which are dispatcher and workers. In the chapter about workers is described how they work, whatever they are doing, how a user can use them and a lot of another things. In the same spirit is carried a chapter about dispatcher, where is also described how a user can work with the dispatcher and how dispatcher manages its workers.

In chapter 10 is shown a diagnostic form of database library in which are shown relevant data about dispatcher, workers, SQL statement execution, or there is also the possibility to handle errors and modify SQL statements or its parameters. In the last chapter of this diploma thesis is stated how were implemented automatical tests of this library and how was the library tested in a production application.

2. Asynchronous access

Asynchronous input or output processing is extensively used in the form of data processing. For example, in writing some data to the hard drive is physically writing on the hard drive very slow compared to their processing with the processor clocked to gigahertz units. In that case is asynchronous access beneficial, because the processor can spend his time for another process meanwhile, the data are physically written to the hard drive. Similar useful use can be found in communications, where data exchange is provided separately. When the program needs to send some data, then it passes this data to a communication thread, which performs this data sending and program can meanwhile perform another action and is not blocked. [24]

2.1. Asynchronous vs Synchronous

The main difference between asynchronous and synchronous access to any I/O operation describes the time flow diagram in *Figure 1*.

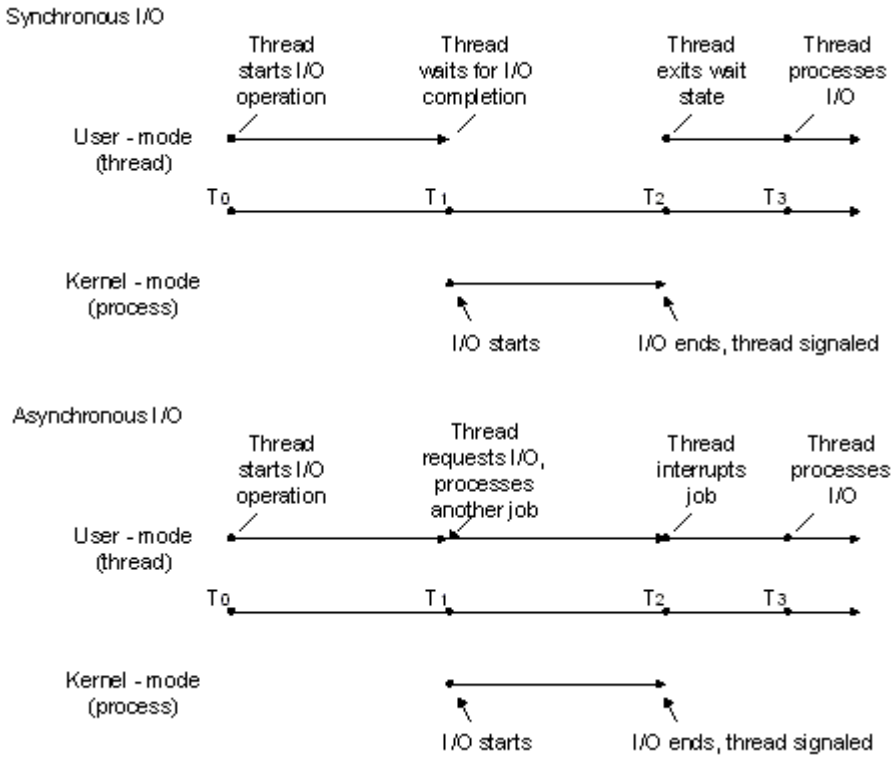

Figure 1 Differences between synchronous and asynchronous access [23]

The main difference is between points T_1 and T_2. In **synchronous** I/O operation user main thread performs a request for some data to the kernel and **waiting** until kernel finish this task and return required data to the main thread. When kernel finishes the task, it notifies about it main thread, and the main thread exits this wait state and processes the data from the kernel. Instead of the synchronous I/O operation, **asynchronous** I/O operation after performing a request for data to the kernel **begins processes the** second task. When kernel have prepared all data, it notifies about it main thread. Main thread interrupts processing the second task and starts processing the result data from the kernel. [23]

This description can be also used as a general description of how applications now work (synchronous) with data from the database and how will be removed unpleasantly freezing by using asynchronous access. The primary task is to remove these periods, where application just waiting until the database returns the required data or perform an appropriate action. This waiting time must be entirely replaced by time, where the main application thread can process another task until the result from the database is prepared. How exactly these actions are performed is already the topic of the subsequent chapters.

3. Firebird database

Firebird is a relational database system, operable on a wide range of operating systems, between them are even the most famous and the most used systems like Windows and Linux. Firebird is based on the foundation of the InterBase database system. Roots of InterBase were beginning in 1976 when Jim Starkey started to work on this database. InterBase had gone a long way with lots of development modification and owner changing until 1994 when it became to the property of Borland Company, which is well known with their other development tools like Delphi or JBuilder. [1]

In 2000 Borland decided to release the source code of beta version InterBase 6.0, which aroused interest in this product in many developers. However, this decision has not long duration, and after a while, Borland decided to end developing open source code and started again developing paid versions behind a closed door. Between these two moments, some independent projects that as Firebird or Yaffil were created from the released InterBase version. Due to this historical moment are available two projects based on InterBase 6.0. today – InterBase by Borland and Firebird, which are more than 95% similarly. [1]

Firebird entirely supports SQL language for database operations. In client application is generally used API called dynamic SQL (DSQL), which allows creating SQL statement dynamically when an application is running. The possibility is also to use the compiler of the encapsulated static language, which permits to insert SQL statements among the commands of programming language. Firebird also provides support for multiuser access to a database with transactions management, triggers, database stored procedures, support for operations with big data, and more. [1]

3.1. Database server and clients

Firebird is based on topology client-server, which means that in topology exist database server, where is also placed database and several client workstations, which operate with the database under the administration of the database server. The most used communication protocol for these operations is TCP/IP but also is possible to use NetBEUI in small networks based on Windows. [1]

Firebird client is whatever application, which provides access to functions from the database server, or to data from the database, for the user. These clients don't have access directly to the database, but they communicate with a database server by requests and responses through the client library. This library, gds32.dll for Windows and gdslib.so for Linux, provide an interface for the application towards database server and must be installed at each client station. Among the essential functions of the library belong creating a connection to the database server and performing database operations. This library is also able to communicate with more database servers at the same time. [1]

Firebird server is a program, which provides service and interface to the database for a client application in a network site. This server **must** be placed at the same computer as the database because this server is only one program, which has direct access to the database. All other applications can get access to the database only by this server. Firebird server can process the request from lots of clients at the same time. These clients can be placed at the same computer as a server or connected to the server by a local area network or internet. Each client has its transaction for isolation

of his operation from another client. The topic of the transaction is detailed described further in this thesis. [1]

For the better understanding of described client-server topology is in *Figure 2* drawn one example. There is one server station, where are placed two databases. On this server station is also installed Firebird server which provides access to each database for applications. On the server station is also installed one client application, which is connected to the Firebird server by local connection and have its transaction. There are also two workstations, where each workstation has installed a client application. Both workstations are connected to the Firebird server, where one of them by the internet and the other by the local area network. Each workstation client application also has its transaction, and all client applications certainly must use a client library for the communication with the Firebird server. [1]

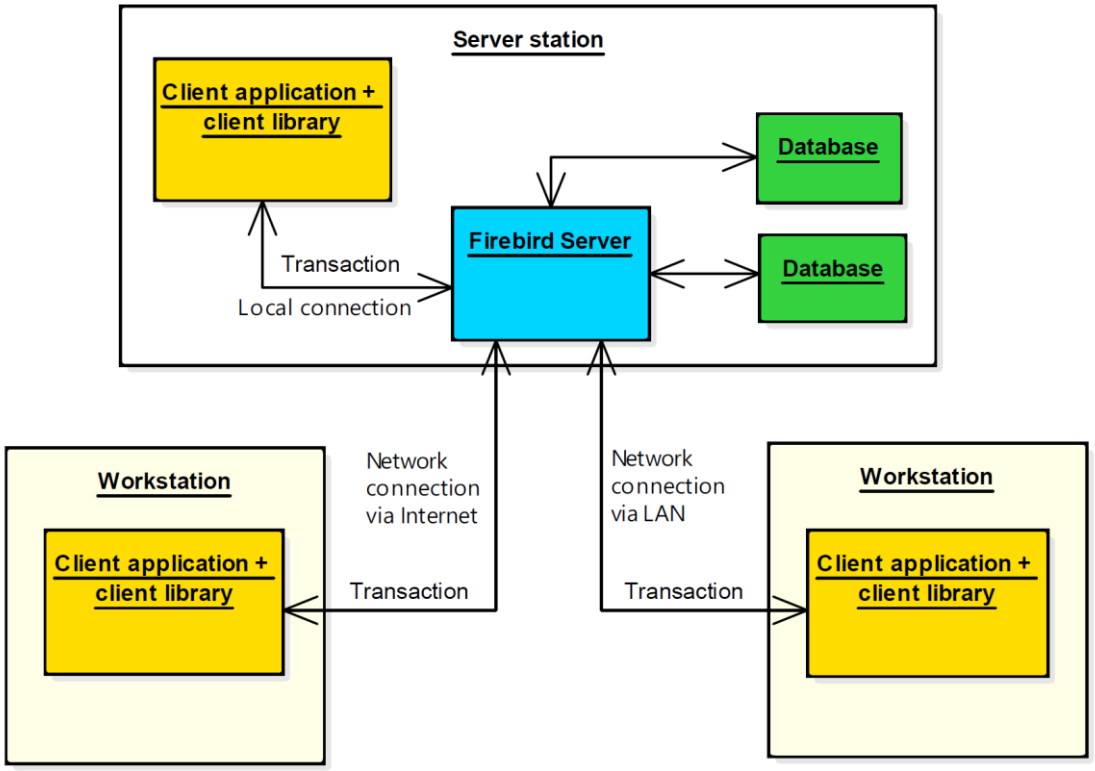

Figure 2 Example of database topology

4. Transactions

Because the aim of this thesis is multithreaded asynchronous access, where each thread has its connection to the Firebird server, then topic about transactions must be very well analysed. The main reason for the transaction analysis is the best optimisation in redistribution SQL statement between individual thread so that they can be executed in parallel form without data conflict between transactions. In this topic is just described what the transactions are, why they are used, and how they are implemented in Firebird. Topic with optimal redistribution SQL statement between individual thread is analysed later in this thesis.

One of the greatest benefits in the client-server database type is transactions. Its main mission is to secure data consistency and provide data integrity in the transition from one consistent state to others. Transactions are not used only for a database system. They can also be used for any dynamic system, where ensure data consistency is required. Philosophy of transactions is based on the assumption, that before transaction start is data in a consistent state. After the transaction start, several operations with the data are performed under the name of the transaction and after performing all operations are changes physically saved to database and data are in another consistent state. Such transactions can be executed several in the same time, and transaction system ensures for them data consistency and prevents mistakes in database data, which can be caused by operations from two transactions with the same data at the same time. [1]

4.1. ACID

ACID is the shortcut or symbol, which is composed of the first letters of 4 main transactions features. These features are atomicity, consistency, isolation, and durability and each of them has its purpose described in this subchapter. [1, 2, 3]

Atomicity is a feature, which ensures the whole processing part of a transaction or not at all. For example, in the move of money between two accounts in the bank **must** be performed deduction from one account and attribution to the second account. The situation that is performed only one of them **must not** exist, either are performed both actions or any of them. Atomicity ensures that if an error appears during transaction processing, then all changes are cancelled, and data are returned to the state before the transaction start, therefore to the last consistent state. This data returning to the state before the transaction start can also be performed by the client using the command **rollback**. [1]

Consistency is a feature, for which are not transactions directly responsible, but they are related to it. Consistency mainly depends on the database model, rules for table columns and relations between tables columns. The server control consistency meanwhile is operated with the data. This control is ruled by direct rules of the database, database model, database triggers, stored procedures and another. The aim is kept the format of data by these rules. [1]

Isolation works with interference between two or more transactions. Its main mission is preventing operations with the same data at the same time by more transactions. Complete prevention of these situations is possible only with transactions serialisation, which means that transactions are performed gradually one by one. This serialisation has but a big negative on the system performance,

which is very slow. Due to this big negative is the use of serialisation practically unthinkable and some methods of synchronisation must ensure isolation. This synchronisation is by SQL standard called **transaction isolation level** and is described in chapter 4.2. [1]

Durability is an easier feature for the understanding from these 4 transaction features. It means that all operations and data changes inside the transactions are durable only when transaction perform all operations and at the end, it confirms these changes by command **commit**. [1]

4.2. Concurrent transaction processing

When two or more transactions are modifying the same data at the same moment, then there may be some problems with data consistency. For example, if one transaction modify data in a database and another transaction modifies the same data before the first transaction its changes approve, then the changes from the first transaction are lost. However, problems may also occur in just reading data from the database. The theory knows two problems related to the concurrent transaction execution and reading data. The first is known as **nonreproducible reading**, where two reads of the same data from the database in one transaction have a different result because another transaction has modified them. The second problem is called **phantom lines**. This problem is like nonreproducible reading, but here is not different data as such, but only their quantity between two reading (less or more records). [1, 2]

For insurance before previously mentioned problems must server manage access to the data for each active transaction. This managing is almost the same as the managing access to the shared medium in the multithread applications. Because the requirement for the best speed and permeability of system are still present, then SQL standard defines so-called **isolation transaction level**, which states different levels, where each level represents a certain degree of compromise between speed and data consistency. This isolation transaction level is a feature of each transaction, and each transaction may have different level. In the next paragraphs are described isolation levels, which are defined by standard SQL92 and its modifications in the Firebird database. [1]

Read uncommitted isolation level permits the highest permeability between all levels. Concurrent data modifications are blocked, but this level enables to read unconfirmed data, which can cause data inconsistency in further processing, which contradicts with ACID rules. Due to this feature is recommended to do not use this level and in Firebird this level does not exist at all. [1]

Read committed level also blocking concurrent data modification, but unlike the previous level, it blocks reading of unconfirmed data. Thought reading of unconfirmed data is not allowed, at this level can still appear problems with nonreproducible reading and phantom lines. This level is very effective for the transaction, which performs just data modification. On the other side is recommended to don’t use this level for large transactions, which processing a large amount of data, or making some changes based on previous data read. In Firebird is this level used exactly in the same way as standard described. [1]

Repeatable read level has the same positives as the previous level and above that guarantees, that read data cannot be modified by another transaction. It prevents before present of nonreproducible read, but phantom lines may still appear. The equivalent for this isolation level in the Firebird is

called a **Snapshot**, but Snapshot doesn't guarantee that read data are not changed by other transaction as in repeatable read. On the other side, Firebirds snapshot guarantee stable data view without fantom lines. [1]

Serializable isolation level is the last level in SQL92 standard which ensures maximum data consistency without any interference between concurrent transactions. This level behaves as if the transaction were executed gradually and because of that there are no problems like a nonreproducible reading or fantom lines. Due to these rules, other transactions can only read data from processed tables in first transactions and its causes speed decreasing of the whole system. According to these features is this level recommend only for the urgent cases. Firebird has also equivalent for this level, which is called **snapshot table stability**, which has the same features as serializable. [1]

4.2.1. Transactions conflicts solution in Firebird

Transaction isolation levels described in the previous subchapter only defines requirements for stability of data for each transaction. When two transactions want to perform some operations with the same data and this operation is not allowed by their isolation level, then arises conflict, which must be solved by the server. Firebird server use for conflicts minimalisation method called **optimistic collision solving**. This optimistic method not preceded collision by data locking as it is in the **pessimistic locking** method. In optimistic method are modified data, or data which must be preserved for repeatable read, just monitored and by that they are protected before changes by other transactions. In the case when some other transaction wants to modify "locked" data, firebird must this request blocked. After blocking this blocked transaction have two common options, either it can wait when lock above data is released, or transaction execution can be interrupted with the fault due to transaction collision. Ordinary is better to use the second option because the blocked transaction can continue when the first transaction, for which it is waiting, ends with rollback only, but in practice, majority transactions end with data confirmation by the commit. For this second option is too possible set by parameter, if the notification for a client about transaction block and interrupt is announced immediately (parameter **NO WAIT**) or is announced after the end of data processing of the first transaction, due to which this block happened (parameter **WAIT**). Use of the parameter WAIT is much better in practice to avoid multiple transactions blocking due to immediately re-execution of blocked transaction. [1]

4.3. Transaction models

In previous subchapters was defined what transactions are, why they are used in the database and how they are implemented in Firebird. Because the diploma thesis aims to develop database library, which can process any amount of SQL statement, then must also be analysed how these statements could be executed for the best optimisation of performance concerning ACID rules. Does it mean that must be known answers for questions like a how optimally manage transactions? Is better to distribute SQL statements between several threads? If yes is better if each thread has its transaction for processing or all threads perform their actions under one transaction? What to do if the database throws an exception due to transaction collision? To find an answer to these questions, in this subchapter are described some known transactions models for client applications.

4.3.1. Flat transactions

The flat transaction is the simplest type of transaction. All modifications in database and SQL statements execution starts after the transaction begins. In most situations, these modifications end with transactions commit, which means that all modifications are physically saved into the database. In some situations, this transaction can end with the transaction rollback command, which means that all changes from transaction begins weren't be stored. The best example for the flat transaction is simple bank transfer, which was being mentioned in the explanation of transactions and *Figure 3* show it. In the transaction must be executed both operations (withdraw and deposit), or any of these operations. Anything else is not allowed. [2, 9]

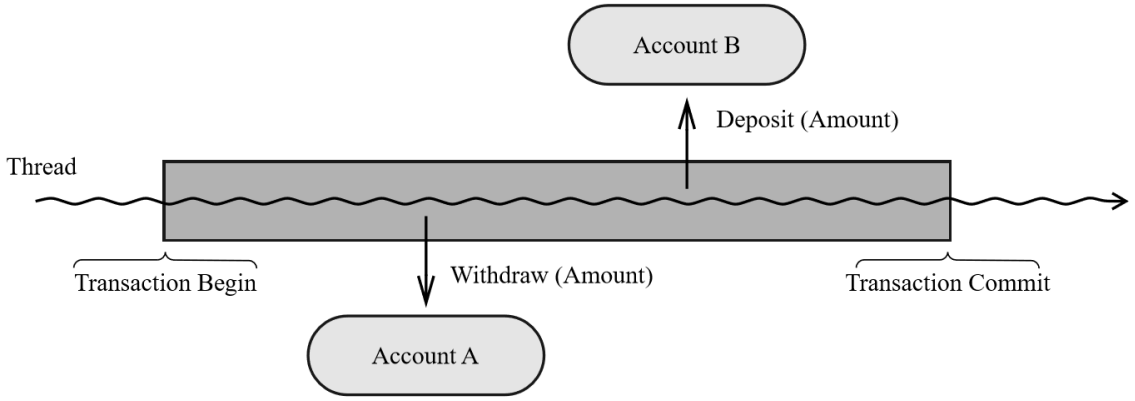

Figure 3 Flat transaction example [9]

The big disadvantage in flat transactions is that in most systems is not implemented exception handling for transaction collision. Instead of that are only used error codes, which inform the user what happened, but the exception is not correctly solved, which is terrible practice. [9]

4.3.2. Flat transactions with savepoints

This type of transaction is almost the same as the previous one, but this type has one advantage. When in the transaction appears some error during execution, then is possible just one option how reacts on that. All operations in a transaction must be rollbacked and only then is possible to start this transaction again. In some short transactions, this option is not very problematic, but if this happened in some larger transaction, then it can be very unpleasantly for system performance. For this problem is there flat transactions with savepoints, where these savepoints can be used in the transaction between some SQL statements and if some error occurs, then is possible to go back to some established savepoint and continue with executing again from this point without aborting the whole transaction. [2, 9, 10]

In *Figure 4* is one example of a flat transaction with savepoints, where is shown one transaction which approaching to two objects. Also, are there established three savepoints. If some error occurs in operation „OpB1", where transaction access to „Transactional Object B", then is possible return back to the „Savepoint 2", and finish all operations with „Transactional Object A" with the commit. [9]

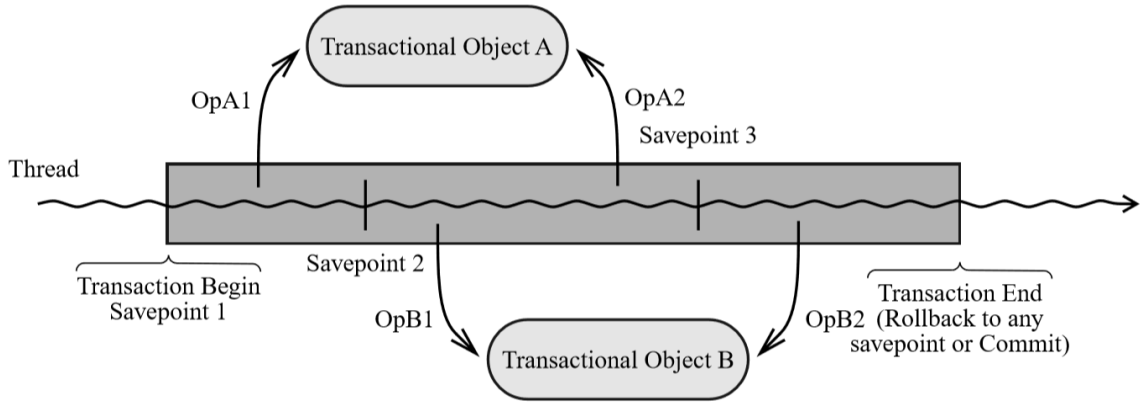

Figure 4 Flat transaction with savepoints example [9]

One interesting thing is that transaction begin also establishes savepoint. The advantage of this behaviour is the possibility to go back to the state when this transaction was started. In that case are all rights for the approached objects kept because the transaction is still alive. [9]

4.3.3. Chained transactions

Chained transactions are very similar types to the flat transactions with savepoints. However, instead of volatile savepoints is there command called chain transaction. This command is like a commit transaction and begins a transaction in one, so it means that previous works are committed, and a new transaction is started, but with the rights of the previous transaction. One example of a chained transaction is in *Figure 5.* [2, 9]

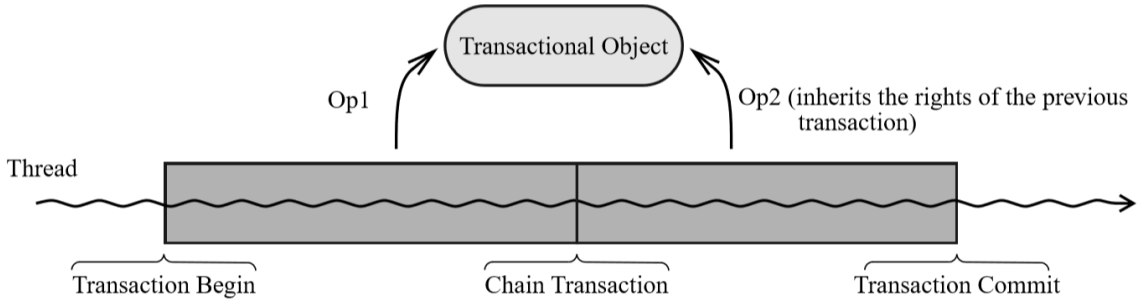

Figure 5 Chained transaction example [9]

4.3.4. Serial nested transactions

Nested transactions are the next extension of the flat transaction model. In this model, a transaction can start subtransactions, which are called nested transactions. The root transaction, at the top of the tree, is called the top-level transaction. The transactions at the bottom of the tree are called flat transactions. Each subtransaction has its predecessor in the tree, which is called a parent and for this parent is subtransaction its child. *Figure 6* shows an example with one top-level transaction and three subtransactions. [2, 9]

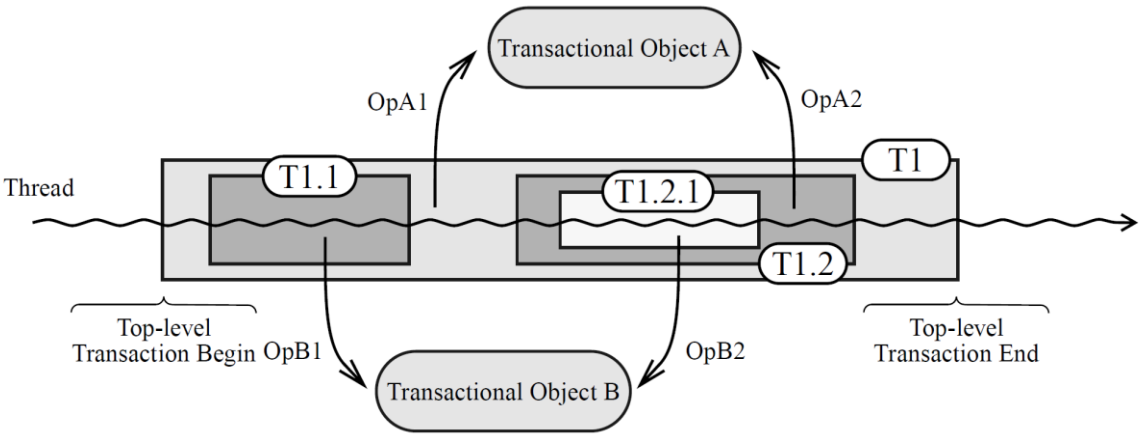

Figure 6 Serial nested transactions example [9]

This transaction model has some specific rules, which must be respected for transaction nesting. The new transaction can be created at any time, and if this new transaction is created in some other transaction, then this new transaction is nested transaction. Nested transaction accesses to specific transactional objects are isolated but concerning its parent and other transaction. Each object, which is held by the parent, is also accessible for the children of the parent. Parent transaction can commit its work only when all its children commit their work. When some nested transaction commits its work, then the result is only visible for its parent. The result of the work is physically stored into the database, and visible to the outside world, only when the top-level transaction is committed. A similar rule applies to transaction abort. If some transaction is aborted then all its children are aborted too, it means that if the top-level transaction is aborted then all inside transactions are aborted too. For implementation must be remembered that transactions at the lowest level of transaction tree are not fully equivalent of the flat transaction concerning ACID rules. They preserve consistency only in their local functionality, they are isolated from other transactions inside and outside the parent transaction, and as was here mentioned, their changes are durable only when the top-level transaction is committed. [2, 9]

4.3.5. Concurrency nested transactions

This transaction model is the first mentioned model, which counting with multithreading in applications. When can some SQL statements be executed concurrently, so why do not execute them concurrently? It is the main idea of concurrency nested transactions, which increases the performance of the system. *Figure 7* is one example of this transaction model. The whole transaction starts as single thread transaction, but after this start is created a second thread. Each thread starts its nested transaction, which is also committed in the same thread. When nested transactions are committed, the second thread is destroyed, and the transaction continues just in one thread and finally commit all transaction. Rules of transaction nesting are the same as in the serial nested transactions, which means that if anything happened in one or other nested transaction, then the top-level transaction can abort all transaction and database still be in consistency state. It means that operations like a withdraw and deposit money between some bank accounts could run concurrently without any problems. [9]

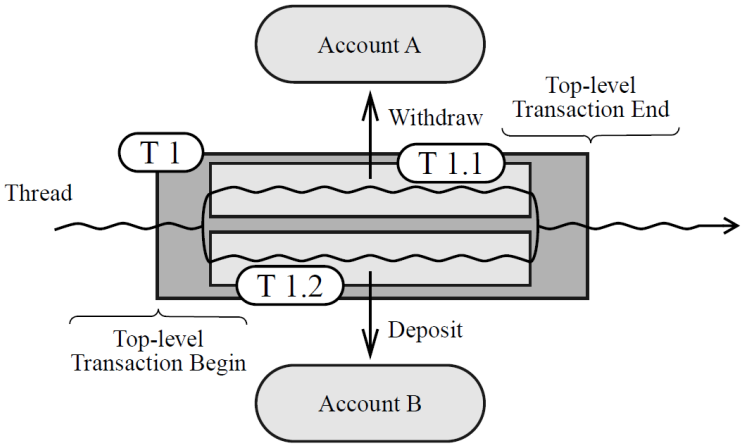

Figure 7 Concurrency nested transactions example [9]

4.3.6. Split transactions

Split transactions were designed to deal with long running transactions. They can physically store to database certain results of transactions processing before they are completely committed or aborted. For this behaviour, they are also using concurrency. They can create other threads with a new transaction meantime they are executing SQL statements, where they pass rights for transactional objects previously processed by them to the new thread. These new threads can process some next SQL commands in parallel with the main transaction, and they can commit all changes which were performed by this secondary thread and by the main thread before it gets the rights of transactional objects to the new thread. The most important thing is that commit of the secondary transaction can be performed completely independent of the main transaction. It means before or after the commit of the main transaction. Is possible to show performed changes to the outside world before transaction which made these changes is committed with these rights passing to another transaction. Next big advantage is that this mechanism may release some transactional objects, for which are waiting next transactions. [9]

Figure 8 shows an example of this type of transaction. After the start of transaction T1, this transaction approaching transactional object A and transactional object B, where performing some changes. After that, this transaction split-off new transaction T2 with a new thread, which gives the rights to transactional object A. It means that transactional object B is from now fully controlled by T2 and T1 has no rights to this object. After some next changes, T2 is successfully committed, which means that changes in OpB1 and OpB2 are committed into the database. After a while is also committed T1 with its changes OpA1 and OpA2. [9]

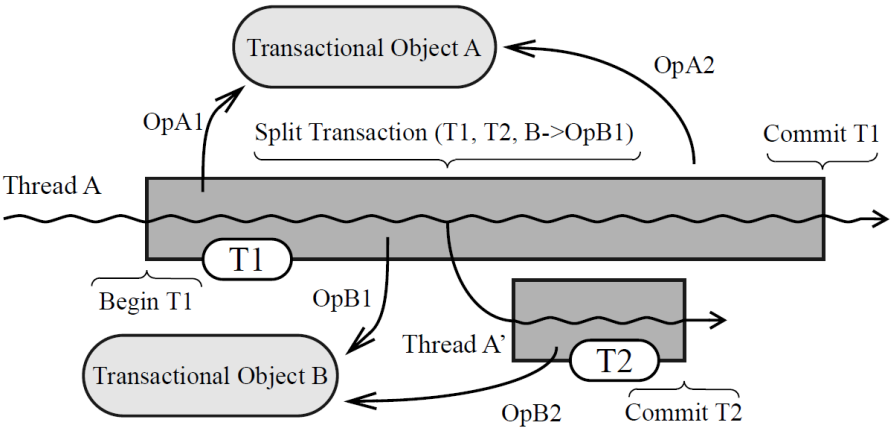

Figure 8 Split transactions example [9]

4.3.7. Joint transactions

This type of transaction is very close to the previous one. Just instead of split-off one transaction into two transactions, in joint transactions are two transactions joined into one. One example is in *Figure 9*, where on the beginning are two threads with two transactions, where each transaction is approaching its external object. When these two transactions are joined into one, then external object B is now in holding of T1, which now may perform next actions with this object. When T1 is committed then are committed all changes by T1 and also changes by T2 before transactions have been joined. [9]

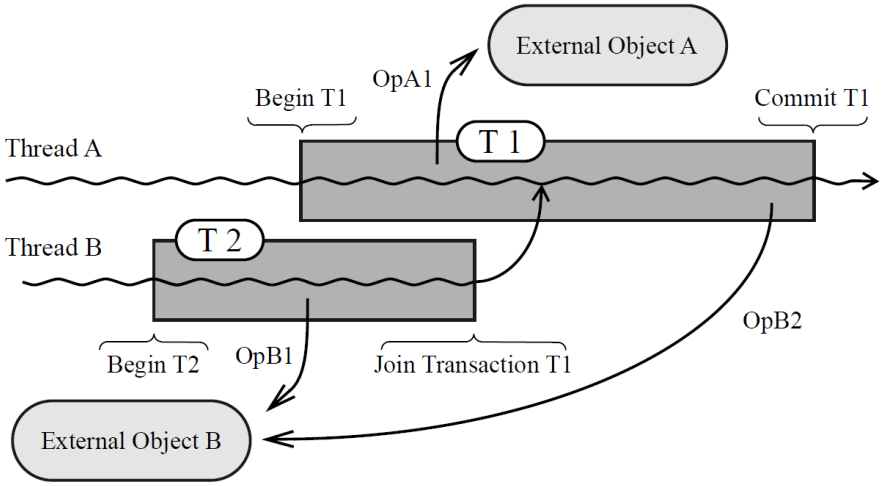

Figure 9 Joint transactions example [9]

4.3.8. Multithreading inside transactions

This model is the next part of transactions models, which use multiple threads for maximising thread performance of the system. This model allows to several running threads get approach to transactional objects on behalf of one transaction as is displayed in *Figure 10*. In this example is transaction started by Thread C and all other threads may freely connect to this transaction when they want. When each thread finishes its work, then leave the transaction and can run further. When the

last thread, which is the participant of the transaction, leaving the transaction then it also commits the transaction with changes of all threads. Performing threads could be freely created or destroyed inside or outside transaction as is displayed on the figure. [9]

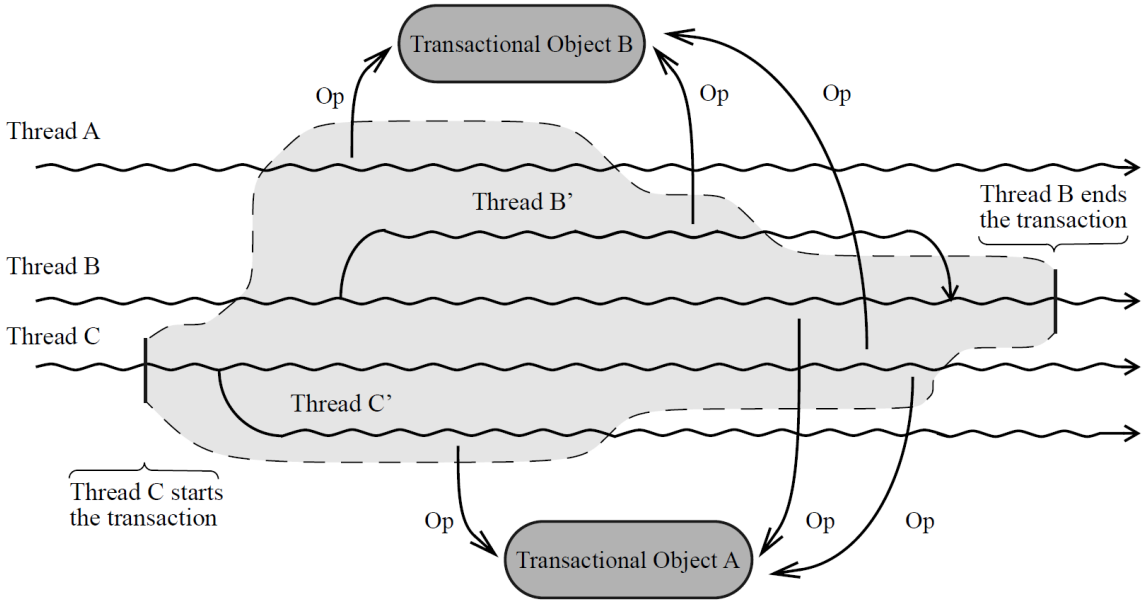

Figure 10 Multithreading inside transaction example [9]

This model is general and is used in many industrial applications. However, the use of this model is quite dangerous because when thread leaving transaction, it does not know if the transaction is committed or rollbacked. It is related to the next problem, where the last thread in the transaction can decide to abort all transaction, without notifying other participants. The next disadvantage is that participants have no idea if they are performing actions in concurrency with another participant and by this data consistency is not guaranteed. [9]

4.3.9. Multithreaded transactions

This transaction model is the next from models, which using multiple threads for executing SQL statements over database data. However, instead of the previous method, here any other thread could not freely enter into a started transaction. Other threads can be only created by a thread, which starts the transaction. Because all threads in the transaction are aware of existing other threads, they could cooperate between themselves, and data could state in consistency. Before transaction commit, all newly created threads must complete its work. At the example in *Figure 11* is there one main thread A which starts a new transaction and after that also creates two new threads for operations. Then each new thread accesses to transactional object A, which is allowed because they are in the same transaction. After a while thread A start a new nested transaction with other two threads. One of these new threads accesses to transactional object A, which is allowed because nested transaction inherits rights from its parent. However, after a while thread A‘ try to access to transactional object A, which is not allowed because this thread is outside the nested transaction, which has now rights for this transactional object, so thread A‘ must wait until the nested transaction is committed. At the end of

the operations are firstly destroyed new threads in a nested transaction, then is committed a nested transaction, and then the same procedure is used for the end of transaction T1. [9]

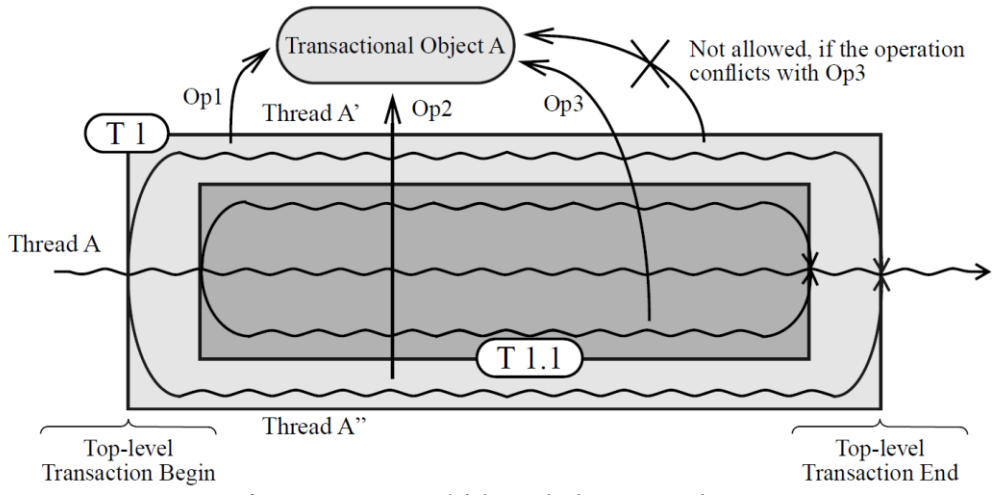

Figure 11 Multithreaded transactions example [9]

4.3.10. Open multithreaded transactions

It is the new transaction model which is designed for requirements of the latest and the most modern systems. This model provides for programmer reliable medium by which he can use concurrency SQL processing without worry that data consistency should be violated. This model could be a beneficial medium for applications that incorporate cooperative and competitive concurrency, dynamic systems which must serve multiple requests in a reliable way and to all applications which work in distributed settings and therefore must deal with various issues. [9]

Before integrating this method into some object-oriented language, first must be analysed if this programming language meets requirement without which this method cannot work. First of all, must this language be able to nest transaction for structuring commands execution and providing fault tolerance. Next, the programming language must support concurrency and if it is possible then use of this concurrency should not be restricted on the inside and outside of transactions. The last requirement is for the ability of exception handling to deal with unexpected events. [9]

Generally, this transaction model secure atomicity and durability from ACID rules. However, data consistency and isolation depend on how the application programmer designs the application. He must take on mind that some error can occur in any thread, so each thread must be able to abort the transaction. Each thread, which finishes its work cannot start performing next actions, or cannot use the data from transactions until it is certain that transaction is committed. Transactional objects must be aware, that some operations could be performed concurrently within one thread and ideally, they should implement some additional control for ensuring data consistency. [9]

This method is closely linked to the multithreaded transaction method, but here is an extension, which allows joining to the transaction for the thread created outside the transaction. This feature is here mainly for real-time systems, where is thread creating and destroying very time consuming and

programmers are trying to avoid this. Create threads in the transaction is also allowed in this model. For **threads** are here just two rules which must be respected: [9]

- A thread created outside of an open multithreaded transaction cannot be terminated inside the transaction.
- A thread created inside an open multithreaded transaction must also be terminated inside the transaction.

Although the threads are joining into transaction separately and performing its specified actions, they should cooperate between themselves, because they could access the same transactional object, which can cause data inconsistency. Each thread which works on behalf of an open multithreaded transaction is called participant. Threads which joined into a transaction are joined participants, and threads which have been spawned in the transaction are spawned participants. In *Figure 12* is one example of an open multithreaded transaction, where threads A, B, C, and D are joined participants and B‘ and C‘ are spawned participants. [9]

In the following bulletins are included the most important rules, which must be respected in the implementation of this method. Rules for **starting** open multithreaded transaction are: [9]

- Whatever thread can start the transaction. The thread which starts the transaction is also the first joined participant
- Participant of the open multithread transaction can start a new transaction which is nested transaction for the open multithread transaction.
- Maximum number of joined participants should be optionally set at the start of the transaction

Rules for **joining** into the open multithreaded transaction: [9]

- The thread can join into transaction whenever it wants, and the transaction is opened.
- The thread can join into open multithread transaction only when it is not in any other transaction. When the thread wants to join into a nested transaction in open multithread transaction, then it first must be a participant of the open multithread transaction.
- The thread which is a participant of the open multithread transaction can spawn a new thread which becomes the spawned thread. This spawned thread can join into the nested transaction and for the nested transaction is the joined thread.

Rules for **concurrency control** in open multithreaded transaction: [9]

- Accesses to transactional objects by participants of the open multithreaded transaction are isolated from all other transactions.
- Accesses to transactional objects by participants of the nested transaction in open multithreaded transaction are isolated from accesses by participants of its parent transaction. This rule is taken over from nested transaction rules.

- Inside open multithreaded transaction could be used techniques to ensuring that to one transactional object accesses only one participant, like a mutual exclusion. This rule is explicitly mentioned for data consistency ensure.

Rules for **ending** open multithreaded transaction: [9]

- All participants after finishing their work vote for the type of transaction end. Possible is only commit or abort.
- The whole transaction is committed only in the case that all participants vote for commit. If only one from participants votes from abort, then the whole transaction is rollbacked.
- After spawned participant vote is this participant immediately terminated
- Joined participants are allowed to leave the transaction only when the transactional result is decided. In case of transaction commit it means, that all participants leave transaction synchronously. In case of transaction abort can participants leave transaction whenever they want.
- If some participant leaves the transaction without a vote, the result of the transaction is automatically rollback.

Correctly behaviour of the open multithreaded transaction according to these mentioned rules is summarised in the example in *Figure 12*. The transaction is started by thread C, which is also the first joined participant of the transaction. After the transaction start, next threads are joining into the transaction, and so they become next joined participants. On this figure is also an example of spawned participant inside transaction and nested transaction. The important thing is that after the vote is joined participants blocked until the result of the whole transaction is stated. Spawned threads are after vote immediately terminated. [9]

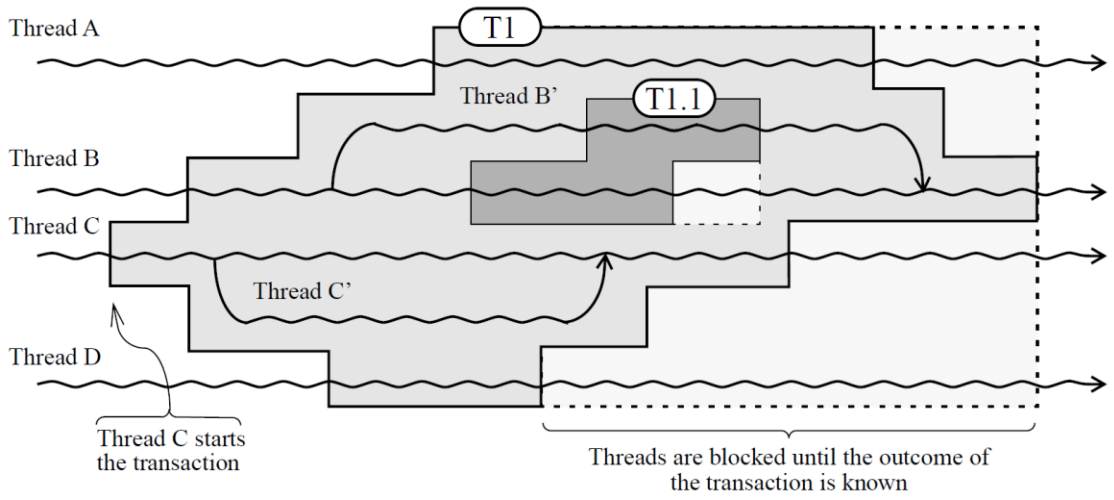

Figure 12 Open multithreaded transactions example. [9]

At this model is also an essential theory around exception handling, when some error occurs. This model distinguishes internal and external exceptions. Internal exceptions are exceptions, which can be successfully handled by transaction participant and they have none globally effect. External

exceptions are exceptions, which participant is not able to handle or solve. These external exceptions eventually cause aborting of the whole transaction. A short example of exception handling is in *Figure 13* where is an open multithreaded transaction with two joined participants. Thread A after joining spawn new spawned participant which make some work, vote for commit and terminates. However, meanwhile this, thread A generates an exception X. This exception is not too much critical, thread A can solve it locally and then also vote for commit. However, meanwhile this also Thread B generate exception Y. Unfortunately, this exception could not be handled locally, and Thread B raise external exception Z. This exception causes is aborting of all transaction. [9]

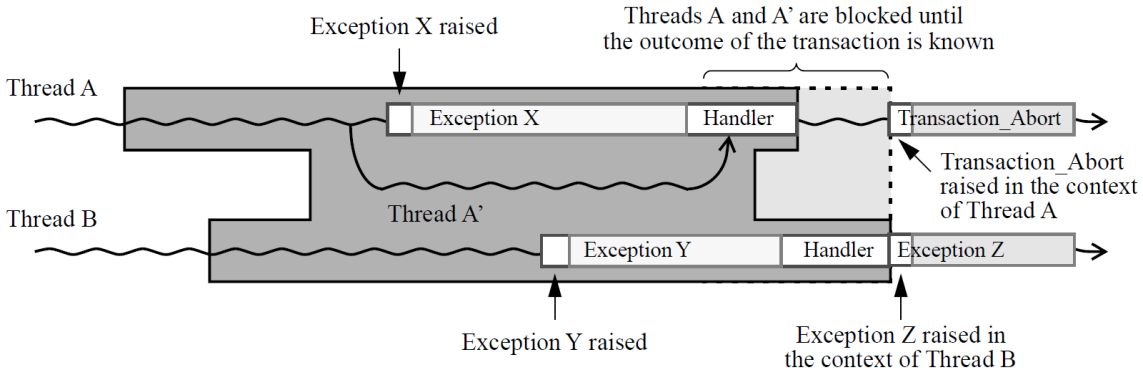

Figure 13 Exception handling in the open multithreaded transaction [9]

In exception handling is also crucial if non-preemptive or preemptive approach is used for external exception handling. In non-preemptive handling are participants informed about transaction aborting after their choice to commit or abort the transaction. In long-running transactions, this approach is very unpleasantly for system performance. On the other side, with a preemptive approach are participants informed about transaction aborting immediately, which frees them from unnecessary work. [9]

5. Delphi

Delphi is a development environment created by Borland company in 1995. Desktop applications with a visual interface and also applications running just in the console is possible to develop in this environment. Application source code in Delphi is Object Pascal, which is object-oriented language based on Turbo Pascal. Delphi is constructed for useful application developing, where prepared components can be used for database connection, internet support, client/server applications, or graphics. [5] Nowadays is also possible in this environment develop applications for web or mobile application and the same source code can be compiled for modern systems like a Microsoft Windows, Linux, macOS, iOS, or Android. [25]

One of the essential things in Delphi are components, which are building elements from which are applications constructed. These components should have visual form and could be seen in the application form, or they only could manage data. Visual components are components like buttons, edits, labels, or tables and data components should be components for connection to the database, access to the internet, or communication components. Each component has its properties and events, where properties could change the visual form, modify component behaviour, or specified setting. In events can programmer react to the specific events which happen. Because components are inherited from a class, so they also have methods, which can be used. For example, regular component class *TComponent* has methods for asynchronous processing, which are described in chapter 5.5.1. [5]

5.1. Delphi high-performance applications

System performance could have many scales, but application users have only one scale. This scale is how fast the system reacts to user commands. It means that if the system reacts to user commands fast enough to his satisfaction, so for the user is this system powerful. When the same powerful system is accelerated for example 5 times, for the user it is not a change. The user notifies improvements only in cases when is increased react time of the system, which was before very slow, has a considerable reaction time, or was always blocked. [4]

In systems are two types of performance. The first type is real system performance, which means how is system fast for example in loading data from the database, or data computing. The second type is the perceived speed, which means that the system responds to user commands very fast. The user perceives it as a system with excellent performance, but all computation is performed at the background, for example in other threads. For each of these two types must be used different techniques on how to increase its speed. For the second type was answer mentioned, the system must be optimally divided into threads, so that one thread can respond to user commands immediately. The algorithm change could increase the first type of speed. [4] When these two types of speed increase are related to this diploma thesis, so the first type seems to be very difficult to implement. The real speed depends on connection conditions to the database and mainly on SQL commands. Both of these things are not in maintain of database library so there must be focused on the second type of speed increase. In performing SQL commands in background threads and letting main thread free for immediate response to operator commands. In other words, by asynchronous SQL execution.

5.1.1. Processes, threads, multithreading

When the operating system starts an application then creates a new process, which has its application code, memory, file handles, device handles, sockets and so on. Each process has at least one thread, which has information about current execution address, CPU registers state or program stack. Each system supports minimally one thread per process, but some modern systems go for better performance and flexibility and support more than one threads per process, where the first thread, created with the process, is the main thread and other threads are called background threads or workers. [4]

In most systems like Windows, Android, or iOS are processes heavy, which means that it is hard to create or destroy them. In the same naming threads are light, so they are created practically immediately. Processes are completely isolated, which means that they do not share its memory and so on with another process and crash of one process do not crash another process. However, the same rules do not apply at threads which means that programmer must carefully protect accesses to shared variables between threads and correctly catch exceptions, otherwise exception from one thread in the process could abort another thread in the same process. [4]

When the operating system has multiple processes and can switch processor time between them, then this operating system is called operating system with multitasking. There are two types of multitasking. The first type is cooperative multitasking, where scheduler assigns processor time to process and depends only on the process when it returns the processor to the scheduler, which can assign it to another process. It is a hazardous type and now is almost historical, because the process may not return the processor to the scheduler, which can cause the fall of the whole system. The second type is the pre-emptive multitasking. The modern systems use this type, where scheduler controls either processor assignment to process and also processor removal from the process. By multitasking could seem that multiple processes are performed in parallel, but physically they are just switched in time. Real multitasking can be achieved only by multiple cores, which is the reality in nowadays computers. Same theory about multitasking with processes also applies for threads. The threads are scheduled, not processes. [4]

5.1.2. Multithreading risks

In multithread application may appear some problems, which do not appear in single thread application. The worst thing of all at these problems is that these problems are difficult to analyse and find out their cause. The only way how to find them is by unit test, which must be written in the best possible way and run several times. [4]

The first problem, which must be avoided by the programmer, is in access to the graphical user interface from background threads. Issues which could raise from this problem are plenty, for example when some threads are trying to manage the user interface, and they access to the same code at the same time, then these circumstances are enough to crash some thread or perform any other unpleasantly subsequence. For avoiding this and similar problems is strongly recommended to never access to user interface from background thread. When is required some operation with a user interface from a background thread, then is strongly recommended use functions *Queue* or *Synchronize*. The *Queue* is a modern function which takes as parameter procedure or anonymous

method and performs it in the main thread. *Synchronize* is the older version, which accepts only a procedure as a function parameter. [4]

Next problem, which must be avoided, is simultaneous reading and writing. One example can be with *TList*, which is a list of same objects, integers, or strings. Imagine the situation, where two threads want to access the list, which has two items of integer, at the same time. One thread wants to delete one item and second thread want to read their values in the for loop. For loop is initialised for two items, but after that second thread delete one item. It means that first round is fine, but the second round throws an exception with *EArgumentOutOfRangeException* because the first thread tried to access an item which now not exists in the list. [4]

The last type of most common issues is a problem with access to the shared variable. One example, which well describes this problem, could be when one thread constantly decreases shared variable in loop and second thread constantly increase the same shared variable. As was mentioned in the previous chapter about multithreading time, when each thread can run and perform its operations, is managed and controlled by a scheduler. Imagine the situation when first thread read the value of the shared variable, for example 100, and is interrupted by a scheduler which gives processor time to a second thread. The second thread also read the value of the shared variable and constantly increase it with the store to memory to value 160, then is interrupted and processor time is backed to the first thread. This thread has road value 100 from previous work, decrease it and store 99 to memory, which is the issue. [4]

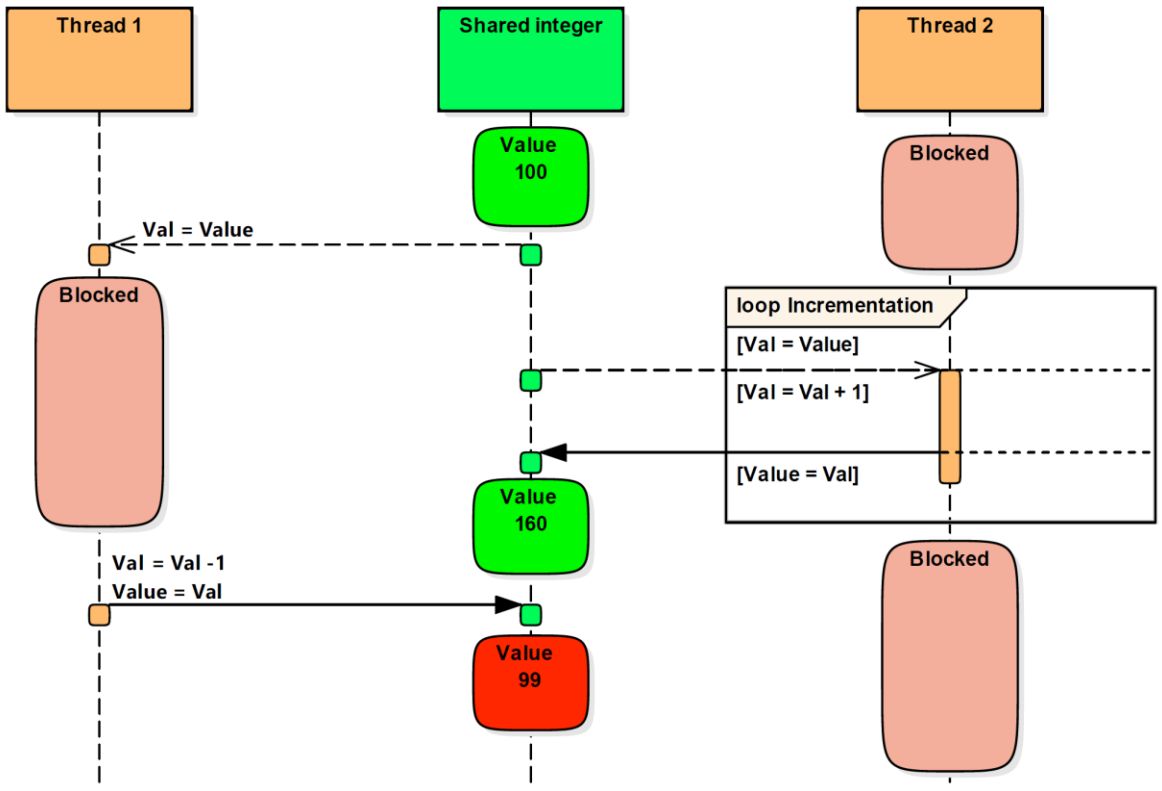

Figure 14 Problem with access to a shared variable

These problems should appear plenty because thread should be interrupted after any atomic operation. Atomic operations depend on processor family (Intel, ARM), and its architecture. [4]

5.1.3. Synchronisation

For avoiding bad circumstances from the previous subchapter is necessary to use some synchronisation. Ordinary, synchronisation is based on locking objects, or a specific part of the code so that only one thread could perform operations with the object in the same time and other threads wait until the first thread finishes work and release the object. [11] This situation is the same as in database transaction read-committed level when a transaction cannot access to specific data, which are now processed in the second transaction until this transaction is committed. [4]

First and the simplest type of synchronisation is *TCriticalSection*. *TCriticalSection* is an object which has two main procedures *Enter* and *Leave*. *Enter* means the start of the critical section and *Leave* means end of the critical section. When the first thread enters the critical section, then it owns the rights to the critical section. When some other thread also wants to enter the critical section, then it must first wait until the first thread finishes work and release critical section to other threads by procedure *Leave*. It provides for application more security in connect to problems from the previous subchapter, but object locking and unlocking is relatively demanding and slows the program. These critical sections objects could be created a lot, which brings one disadvantage for the programmer, who must avoid unpleasant circumstance connected to critical sections, which is called deadlock. Deadlock is the situation *when thread 1 acquires critical section A and then tries to acquire critical section B while in the meantime thread 2 acquires critical section B and then tries to acquire critical section A*. *Figure 15* shows this situation. Deadlock always means system freeze and is not special just for critical section. It could also happen in another synchronisation mechanism. [3, 4, 12]

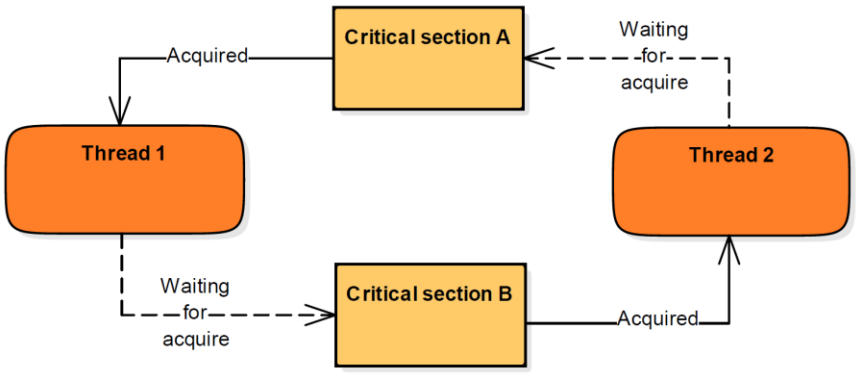

Figure 15 Deadlock

Next mechanism for locking critical sections is *TMutex*. This mechanism has almost the same functionality as *TCriticalSection*, but *TCriticalSection* could be used only in one application, instead of *TMutex* which could also be used between more applications. It is achieved by special name which belongs to a mutex, so when two or more applications create mutexes with the same name, then they could protect access to the shared variable, for example, to file on disk. Next advantage of *TMutex* is its function *WaitFor* with a timeout parameter, by which is tried to acquire critical section in time until the timeout. When is not critical section acquired until timeout, then *WaitFor* returns *WrTimeout*

and continue. *TMutex* also has one disadvantage, which is that it is almost 50 times slower than *TCriticalSection*. [4]

TSemaphore is the next part of the synchronisation object. This object allows access to some part of the code for more than one thread. *TSemaphore* has its limits of maximal threads which can access to the same section. When some thread call *Acquire* and coming to the section, then is semaphore value decreased by one. When thread leaving section by *Release*, then is semaphore value also increased by one. When some thread tries to call *Acquire* and semaphore has value zero, then this thread is blocked and must wait until semaphore value is increased. Semaphore has the same disadvantage with slow performance, but they can also be used for synchronisation between more applications. [3, 4]

Next synchronisation object for critical sections is *TMonitor*. This object has key functions *Enter* and *Leave* for entering or leaving the critical section. The main difference between *TMonitor* and *TCriticalSection* is that *TMonitor* could lock directly objects, which could be protected, and is no needed to create special objects for locking. *TMonitor* also has a good advantage in speed, which is faster about four times than *TCriticalSection*. The reason why is *TMonitor* faster is that it uses spinlock. When *TCriticalSection* tries to enter to critical section object which is locked, then the operating system puts this thread to sleep and wakes it up when the lock is released. When is used spinlock, then is perceived that critical section is concise and at the moment will be released, so the thread is not going to sleep, but stay in some loop where continuously asking for the locked object. [4]

In Delphi is directly synchronisation object *TSpinlock*, which can be used for locking. *TSpinlock* also has methods *Enter* and *Leave* as *TCriticalSection* and moreover has function *TryEnter*, which has the same behaviour as *WaitFor* in *TMutex*. One *TSpinlock* disadvantage is that there is not enabled re-entering. It means that when some thread enters to critical section and before exit try to enter again, then it throws an exception. For interest, *TLightWeightSemaphore* is the next Delphi synchronisation object, which has the same behaviour as classic *TSemaphore*, but here is used spinning for better performance. [4]

5.1.4. Communication between workers and the main thread

Communication between threads can completely replace problems with synchronisation of access to shared variables, so communication technique is more recommended to use but is harder for implementation. The main idea is that data is shared between threads by communication instead of their sharing by variables and here are mentioned some techniques how to do that. [4]

The first type of communication is by windows message. This method uses sending a message from workers, which are captured and processed in the main thread. This sending is performed by procedure *PostMessage*, which two main parameters are message name and value. In the main thread then must be created a procedure for catching this type of message, where the value is processed. [4]

Following resources are not directly for communication between workers and main thread, but they are used for executing some part of code from a worker in the main thread. These resources are *Synchronize* and *Queue*. *Synchronize* accept as parameters *TThread* object and anonymous method

which should be performed in the main thread. When is *Synchronize* called, then is background worker put into sleep until the main thread finishes this job. After job finish is background thread awaked and can continue with its work. *Queue* works completely same as *Synchronize,* also has the same parameters but has one difference that background thread is not blocked when the *Queue* is called, so the worker can immediately continue with its work. [4]

The last method which can be used for communication is *Polling*. In this method, when the worker wants to send some data to the main thread, then it just inserts this data into *TLockedList<T>* or *TThreadedQueue<T>*. This object is continuously checked in main thread timer, and when there is some new item, the main thread performs appropriate action with it. [4]

5.2. TThread and thread pooling

TThread is a basic class for using threads in applications. One disadvantage is that programmers also use this class instead of some parallel pattern which makes the code hard to understand, debug and so on. Out of that can implementation of *TThread* still be readable and very effective. Heart of *TThread* is a procedure, which must be overridden in the descendant with code which should be executed in this thread. *Execute* is called automatically after thread creation, but a parameter can disable this automation in the constructor. Method *Suspend* can pause thread execution, and method *Resume* can start it again, but is strongly recommended do not use this technique, because the thread can be stopped in any part of the execution. By that is a better way to stop thread with method *Terminate* which set up *Terminated* flag, which must be controlled in procedure *Execute* and then thread can be correctly stopped and after that also destroyed. If thread finished its execution can be checked by the *WaitFor* method. [4, 17]

As was mentioned in chapter 5.1.1, thread creation is considerably less demanding than process creation, but in a high-performance application, where each millisecond is count, is also thread creation and destruction demanding. For this purpose, is there thread pooling and *TThreadPool* class, which is also used in tasks or parallel patterns. This thread pool is the storage of inactive threads which are not destroyed and could be immediately reused for new work. When the program runs the first task, then is created a new thread in the thread pool, which after task finish stay in thread pool in the inactive state. When the program runs the second task, then this task is assigned to inactive thread from the thread pool. In case that first thread is still active, then thread pool creates a new thread. [4]

5.3. Tasks

The task is a modern instrument for parallel processing. Difference between thread and task is that thread is part of a system which allows executing some works in parallel form continuously, but the Task is just part of the code, which should run in parallel form. In other words, in thread programmer specify what and how something is executed, otherwise in task programmer just specified what should be executed in parallel form, anything else is not depending on him. [4]

TTask constructor accepts as parameter procedure or anonymous method and returns a reference to interface to this task (*ITask*). After creation can be this task started with the *Start* method or creation,

and start can be merged into one function which is called *Run*. After that, the task can be monitored or managed over the interface returned by the *Create* method. [4, 18]

5.4. Parallel patterns

Parallel patterns are a modern extension of all resources mentioned in previous chapters. Patterns have been developed for effective work with a parallel world and code clarity. [4]

The first pattern is *Async/Await*, which run code at the *Async* part in background task and after that run code at *Await* part in the main thread. This pattern has the advantage that when the code in *Async* is executed, the main thread is responsive and is not freeze. *Async/Await* is the simplest pattern for use in code. [4]

Next pattern is called *Join*. This pattern can start multiple tasks at the same time. *Join* is a method, which parameters are procedures for a run in the background. *Join* returns the same instance of *ITask* as a *TTask* Create with whom is possible do the same operation or monitoring of running tasks. This pattern is currently not working correctly, because it does not start enough threads for procedures. It means that when is required to start two procedures in the *Join*, that both run in the same thread. [4]

Future is the next pattern, which is special for functions because it counts that there is the return value. *Future* is based on *TTask*, so doing similar operations or monitoring running function in the background is possible. As a process for getting the result of the function to the main thread can be selected whichever from the part about communication. *Future* can also start normal function or anonymous function as *TTask*. [4]

With the *Future* the list of patterns is not complete. In Delphi are moreover patterns like *Parallel For*, *Pipeline*, or *Stages*. These patterns are little complicated compared to the previous one and is not required to describe them here. However, ordinary *Parallel For* is constructed for running each iteration of for loop in its thread, so it is some multithreaded version of the classic for loop. *Pipeline* with *Stages* is a technique for the situation, where some job could not run in parallel, but it is possible to split it into various independent stages. The second stage could work with the first instance, and meanwhile, the first stage can prepare the second instance for the second stage. [4]

5.5. Current use of asynchronous access in Delphi

In Delphi programming language now exist some components, which have between its features techniques for asynchronous operations. Let's take an inventory about these techniques, which either can be very useful for implementation of this thesis with its all functionality or at least can be a perfect inspiration. At the first point is analysed class *TComponent*, which is the basis for all Delphi components. Moreover, at the second point is analysed FireDAC library, which is the library for database access, which has the options for asynchronous SQL processing. Based on the previous sentence may seem that this thesis is almost done, but FireDAC has some negatives, which are here described.

5.5.1. TComponent

TComponent is one of the base Delphi classes, from which are inherited all components. In this class is implemented some support for asynchronous calling and method processing. This support is by Embarcadero called *Asynchronous Programming Library*. [19] The essential point of this library is function *BeginInvoke*. This function is overloaded in various form, but the fundamental parameter of all versions is *AProc* or *AFunc*. [20] These parameters are in general some functions, procedures, or anonymous methods, which should run in asynchronous mode. This function returns *IAsyncResult*, which is interface for diagnostic of this asynchronous calling. For example, it has property *IsCompleted*, which indicates if this calling was wholly done or still running. All functionality of this interface is implemented in class *TBaseAsyncResult*, but in *TComponent* it has lots of descendants, where is implemented some detailed functionality almost for each type of *AProc* or *AFunc*. However, in *TBaseAsyncResult* is not implemented only support for diagnostic, there are also methods for asynchronous performance, where some methods are implemented here, and some methods are virtual and must be implemented in the descendant. The most important procedure, which must be mentioned, is *AsyncDispatch*, where should be implemented whatever must be performed in this asynchronous calling. In *TBaseAsyncResult* is this procedure just virtual and descendants of this procedure in *TComponent* call *AProc* or *AFunc*.

It was some little mention of the most important classes and its methods for asynchronous processing at *TComponent*. Now, must be analysed, if the behaviour of this processing is equal to this, which describes chapter 2.1. For testing of all this theory was implemented one test example. The answer for the previous question is **no**, because the task, which should be performed asynchronously in the second thread, is called in the main thread. This fact proves *Figure 16*, where is shown call stack from the point, where was being clicked the button, to point where procedure, which should run asynchronously, has begun to be performed.

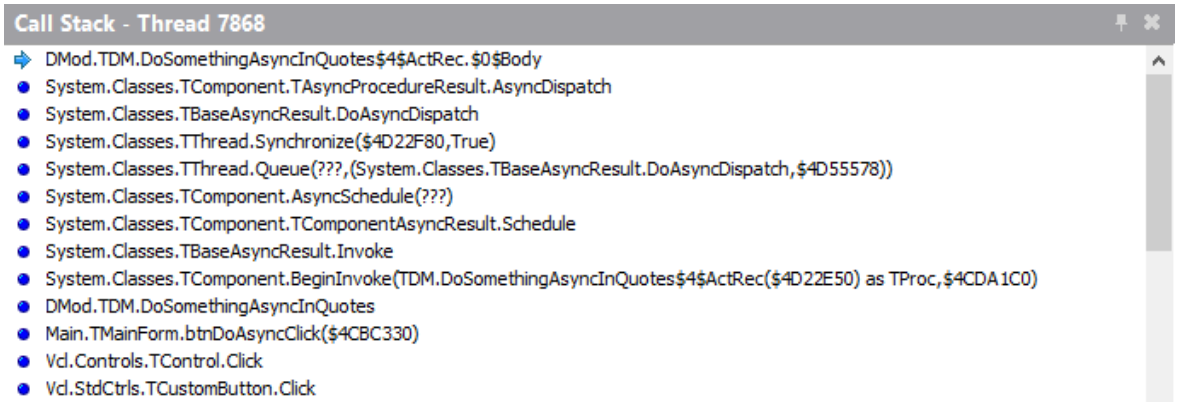

Figure 16 Call stack for calling the asynchronous procedure without any modification

The most important rows are these, where is called *TThread.Queue* and *TThread.Synchronize*. Function *Queue* theoretically causes a call of specifies method asynchronously in the main thread. It means, that specified method runs in main thread asynchronously and calling thread can run further. This behaviour is valid just in that case when *Queue* is called in some other thread, but in the case in *Figure 16* is *Queue* called in the main thread, so it means, that this call is performed entirely

synchronously in the main thread. Procedure *Synchronize* confirms the calling in the main thread. [21]

This fact about the running method in the main thread is too mentioned in the overview of *Asynchronous Programming library*. Also, is here mentioned that when is required different behaviour, in this case method performing in other thread, then is required to override *TControl* function *AsyncSchedule*, where can be implemented self-made functionality for assign task to another thread. [19] This theory was also tried, where for this purpose was implemented a new class, inherited from *TControl*, which override procedure *AsyncSchedule* from *TControl*. In this procedure is the instance of *TBaseAsyncResult* just assigned to the second thread to processing. This cause the correct behaviour. It means that the main thread can run further, and the method is processed in the second thread.

5.5.2. FireDAC

FireDAC is a multi-device library for data access to the database. With this library is possible to connect to various databases which are displayed, with all FireDAC topology, in *Figure 17*. FireDAC provides a flexible engine for data access including data set and data set descendants, which are capable of performing a fast database operation. These data sets also providing functionality for fast data operations like sorting, filtering, performing SQL queries against data sets and more. Between next positive features, which must be mentioned, must belong abstraction layer between all databases, automatic connection recovery and support for asynchronous database operations. [16]

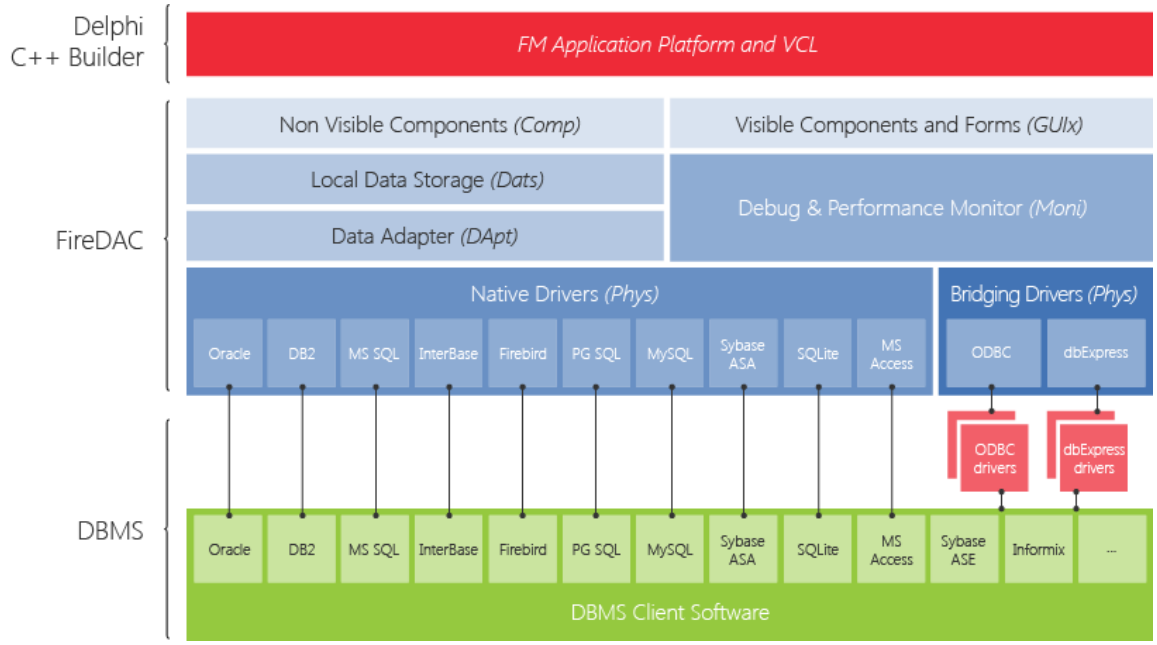

Figure 17 FireDAC topology

When is required to process SQL statement asynchronously, then must be set property *ResourceOptions.CmdExecMode* in *TFDQuery* to *amAsync*. Then after calling procedure *Open* or *Execute* is SQL statement processed in the second thread and main thread can perform some another task. After finish database operations is called event *AfterOpen*, or *AfterExecute*, which is called in

the main thread, and as the parameter has data set with the result data of SQL statement and this data can be here processed. [22] This process is also displayed in *Figure 18*.

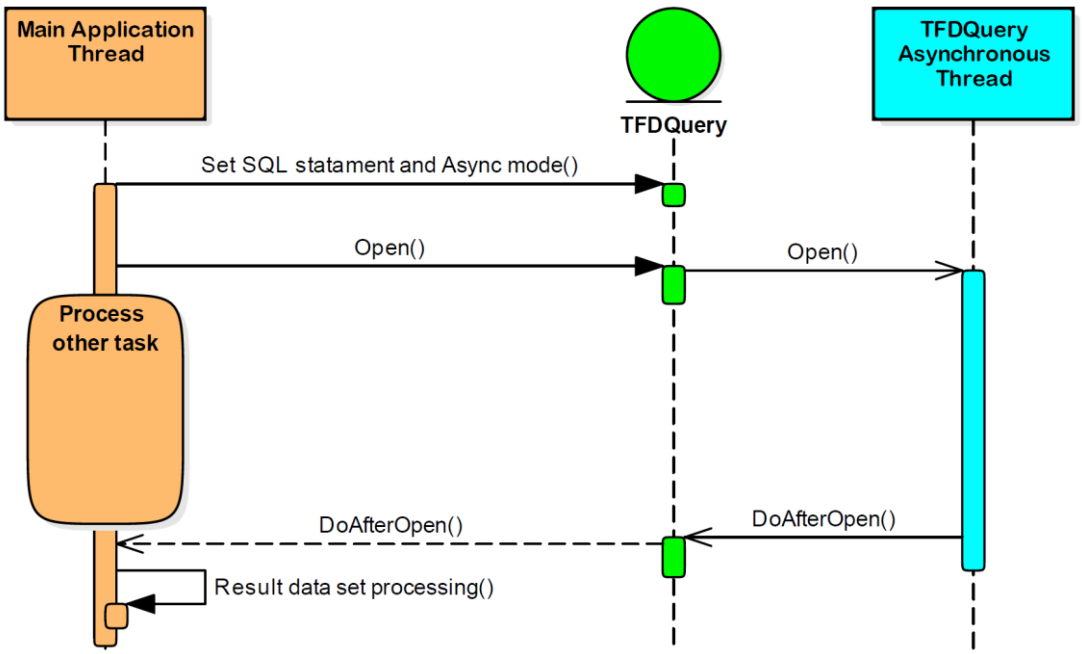

Figure 18 Asynchronous processing in FireDAC

FireDAC is undoubtedly a perfect library for database access with all its advantages mentioned in this chapter. However, on the other side has some disadvantages, which must also be mentioned. First of all, it is not a very good "interface" for using the asynchronous call in the project. If in some part of the code is required to get data from database asynchronously and process them, then must be set SQL statement, created a special class procedure, which must be assigned to After*Open* event and after that can be called *Open* for execution. This process can be summarised into one function, where the procedure for result processing can have a form of the anonymous method, so there is no needed to create a special class function for each SQL statement result processing.

Next disadvantage can be seen in procedures *Open* and *Execute*. Both are for executing SQL statements, but *Open* is for SQL statement, which returns some data, for example ordinary SELECT and procedure *Execute* must be used for SQL statement, which returns any data, for example ordinary UPDATE or DELETE. If in code is some mistake and for SELECT SQL statement is called *Execute*, then FireDAC raises an exception. It can be little unpleasantly at project implementation because the programmer must be cautious if he should call *Open* or *Execute* and if he assigned a procedure to correct event. Moreover, procedures for *AfterOpen* and *AfterExecute* events are little different and cannot be used one for another.

The last disadvantage is that FireDAC does not support multithread asynchronous SQL processing. Embarcadero this fact mentions in the documentation and offers some good options for implementation of this functionality like using a dedicated connection for each query or create its threads and execute the database tasks in these threads. [22]

6. Software testing

Software tests ordinary estimating software quality about problems which can appear. For customers are several risks in each software like data lost, fails of operation, or that they spend more money than software saves for them. All these risks can be eliminated only by proper software tests. Another task for software tests is also that software meets customer requirements. It means that software is not too slow, unreliable, or on the other side also too much complicated according to customer requirements. For these problems are here two essential definitions, which must be included in the testing process. First is validation, which means that software doing what was ordered by customer exactly and second is verification, which is ensuring that system does not crash, all functions working correctly and provide accurate results, or that failures are correctly handled. During these verification tests must also be checked if the code is secure and does not have vulnerabilities, which could be exploited by attackers when they attack a system. In the software world is strongly recommended to make tests because they can reveal problems before software is deployed to the customer, which is cheaper in the long term. Systems with automated tests also ensure that other changes or new functionality do not have a bad influence on existing functionality. [6, 7]

Tester has three possibilities how he can look on a test some functionality. The first option is called Black-box testing. In this situation tester does not know anything about how functionality is implemented, he only knows the inputs and outputs, so he is trying different values pass on input and controls if the outputs of the function are correct. The second option is called White-box, where tester also knows how the function is implemented, he has full access to the source code, so he should test every possible passage of function, and if something unexpected happens, he can analyse what happened. [7, 8, 13] Last option is called Grey-box. This option is something between the previous two. [13, 14, 15] In Grey-Box testing tester knows inputs, outputs and knows internal structure as in White-Box, but the difference is that he does not have access to the source code. [15]

6.1. Manual tests vs automatic tests

Manual tests are tests which are entirely based on the tester. It means that tester follows some script or manual what and how testing and he performs these tests. Automatic test means that this manual or script is transferred to the manual for computer, which performs these tests automatically. In automated tests should be checked all software aspects and return values of functions like in manual tests. In *Table 1* and *Table 2* are compared to the advantages and disadvantages of manual and automatic tests. [7, 8]

If someone considers for which type of tests decide, then he must consider all the advantages and disadvantages of these types. Generally, are preferable automated tests, where is a significant advantage that they are swift, and they can run several times a day, so they can immediately stop programmer in implementing some new things, which could disrupt previous functionality. Almost all modern companies use some compromise of these two types. Most of the tests they are trying to automate, but also sometimes they perform the manual test because automated tests could not reveal all issues and mainly is very hard to reveal issues at the graphical user interface. By this is for companies hazardous to rely only on the automatic test. Solely on automated tests can be relied only

upon in cases that software has not directly for users (for example it performs just data operations) and automated tests are at an outstanding level. [7, 8]

Table 1 Advantages and disadvantages of manual testing [7]

Advantages	Disadvantages
Simple and straightforward	Boring
Cheap	Often not repeatable
Easy to set up	Something can be impossible to test manually
No additional software required	The tester can also make a mistake
Extremely flexible	Time and resource consuming
Testing software like customer	Limits to the only black box or grey box tests
Possible to catch issues which automatic test cannot	

Table 2 Advantages and disadvantages of automatic testing [7]

Advantages	Disadvantages
No chance to human mistake	Extra time for initial setting
Extremely fast	Very hard to catch issues in the user interface
Easy to execute, after first set up	Required knowledge about automatic tests, additional tools, and frameworks
Easily repeatable	Required more skilled stuff
Simple to analyse the process	Testing just implemented the test.
Less resource consuming	Some tests should be unnecessary after time, which often can recognise only human
Ideal for white-box testing	
Additional tests are not too much time-consuming in the long term	

6.2. Unit tests

Unit tests are special automatic tests, which testing the smallest functionality about which black-box tester or users does not know that it exists. These tests can test some functions or procedures in class at the deepest level. Tests are often constructed for testing every possible passage of function with various data. The aim is that function has correct outputs and can correctly handle unexpected inputs. These tests are often created directly by the developer who is familiar with source code and knows every passage and possible problems, which can appear. Unit tests are a perfect strategy to catch problems at low levels. [7] Unit tests have the same advantages and disadvantages as automated tests from the previous subchapter. Also, there is the fact that is practically impossible to test graphical user interface automatically but must be remembered, that there are always displayed data provided from some classes or computation and for ensuring that these data are correct, are unit tests the right way. [7]

7. The topology of database library

It is the first chapter which is related to the solution of the diploma thesis. In the first subchapter is whole problematic divided into functional blocks, which must perform some specified functionality and by this is outlined the solution and general topology of the developed library. In next subchapter is described the class diagram of database library, where are outlined only general things, because the detailed description is in next chapters, where is complete functionality and problems, which appeared during the development of this database library, described in more detail. Concretely in chapter 8 is described how workers work, how they execute users SQL statements, how can the user control them through interface and more. Moreover, further in chapter 9 is described dispatcher, where is described how can the user use this library for executing SQL statements asynchronously.

7.1. Decomposition of the whole problematic into functional components

Generally, the diploma thesis aims to have some component, which can perform SQL statement asynchronously. This component must be able to perform several SQL statements in a parallel form, which means that have more threads which can do this job is necessary. The component must be as much as possible user-friendly, which means that the programmer must have as few options as possible to make some mistake. The component must also count with the possibility that final application is multithread, so there must correctly solved multithreading inside of the component, which means that must be eliminated all possible issues which can appear in multithreaded applications.

It is not the final list of tasks, which the library must handle, but those are the main tasks, without which is this library unusable for practice. According to this list was designed the main topology of database library, which shown *Figure 19*. For the developer is as the first available level for asynchronous execution component called dispatcher. By dispatcher is developer able to add some command into commands list, which is then executed by a worker according to rules described further in the thesis, or he can lock some worker for the primary mode of SQL execution. This primary mode works in a way that when a developer asks the dispatcher for some worker for asynchronous SQL execution, then dispatcher according to some rules described in the next chapters give to developer one locked worker. Then the developer can execute by this worker one or several SQL statements. When he finishes the execution, then the worker is released, and it can do some other work as executing SQL commands, or it waits for the next lock and SQL execution. This scenario is just for illustration. Whole problematics about worker states, locking workers, their run, SQL execution and result processing, describe the next chapters about workers and dispatcher.

From *Figure 19* it is possible to see that dispatcher has several workers at his disposal, where everyone can have a different job, and they can work in parallel. These workers are threads, which are responsible for executing SQL statements. They are living their own lives and wait on the job which will dispatcher assign to them. Each worker have its transaction and database connection, thanks to which they can execute SQL statements concurrently.

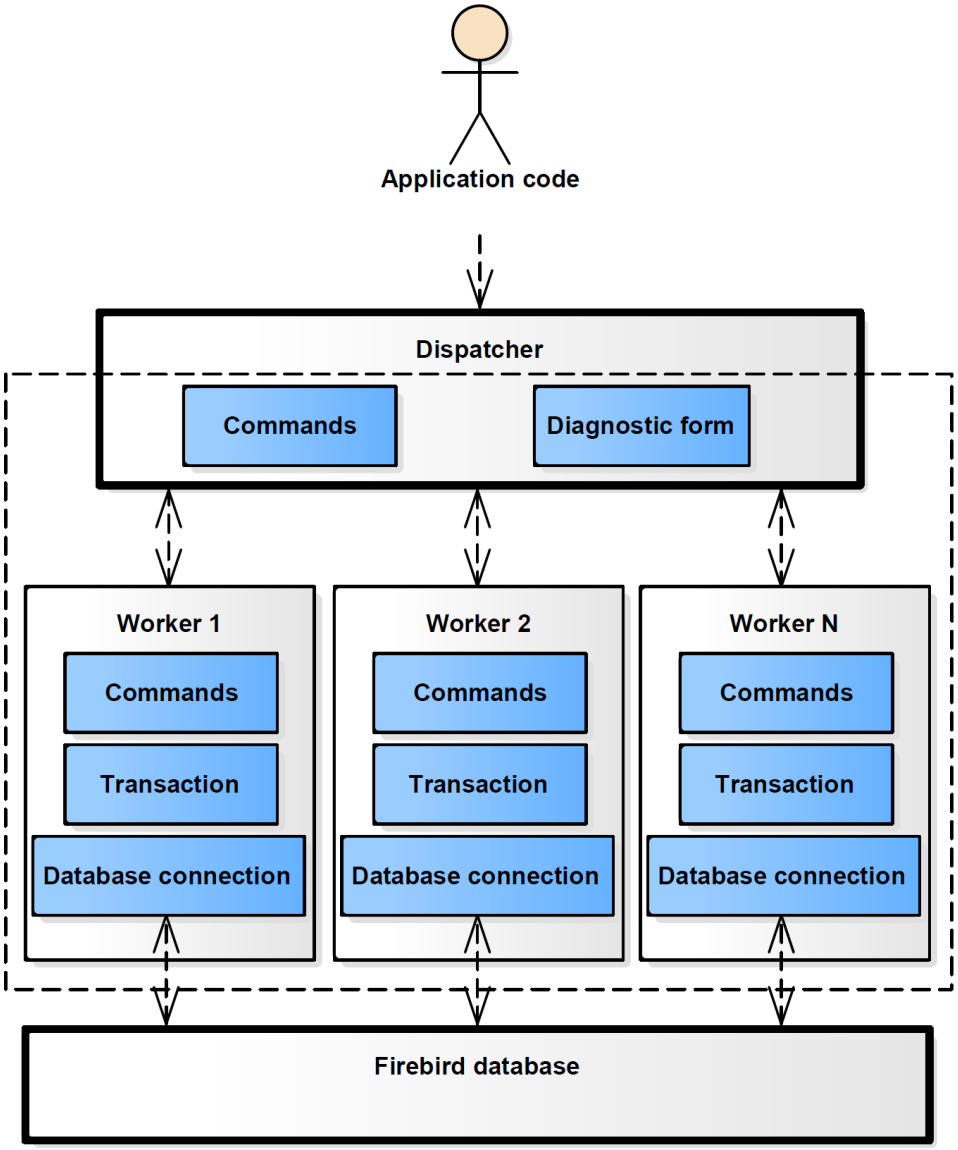

Figure 19 Topology of database library

7.2. Class diagram analysis

According to designed topology, described in the previous subchapter, was developed database library whose class diagram shown *Figure 20*. Here are two bigger classes, which represent the implementation of the dispatcher (*TASDispatcher*) and worker (*TASWorker*). Worker implements interface *IASWorker*, which is forwarded as a return value when is required to lock worker for SQL execution. The next classes are general classes which are used for establishing new commands for workers, SQL testing, or some auxiliary lists. It was the whole introduction to the implementation of the database library. In next chapters are these topics described more in detail with all the pitfalls encountered during the solution.

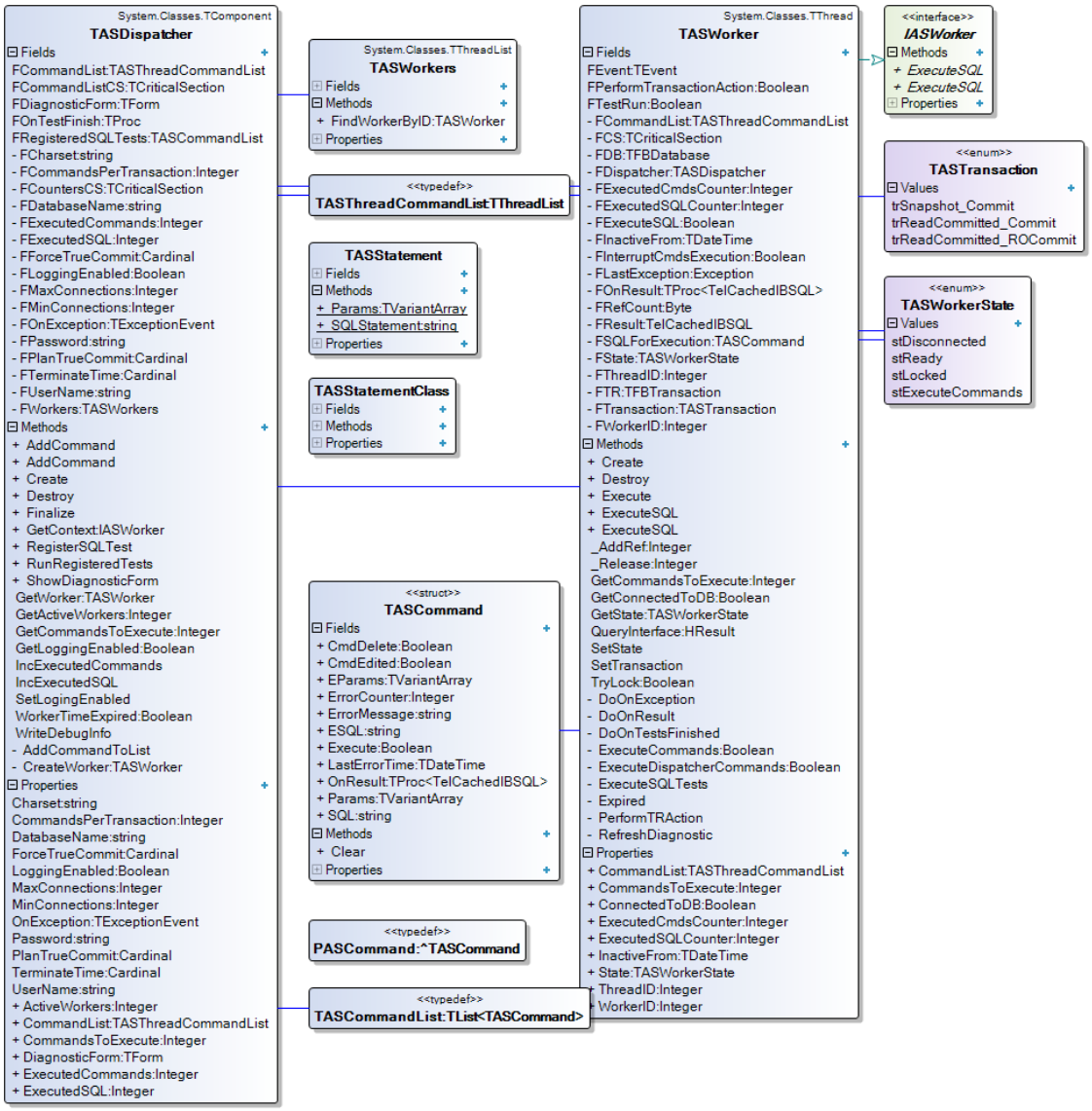

Figure 20 Class diagram of the database library

8. Database workers

Workers are an essential part of this database library because they are performing all the work around executing SQL statements asynchronously. When they are created, or what and when they are doing, depends on the dispatcher, which partially manages them, but about this topic is the next chapter. This chapter is clearly about workers and class *TASWorker*. Here is described how they work, whatever they do and whole their functionality.

8.1. Worker states

Figure 21 shows the worker state diagram with its all possible states. After creation has worker default state *Disconnected* and begins to open a connection to the database. When the connection has been successfully established, then worker changing its state into state *Ready* which means that it is waiting for some task to processing. In this state, worker waiting for some event and periodically testing the connection to the database and in case that it was interrupted, so worker changes its state back to *Disconnected* and trying to re-establish this connection. When is required to lock worker for SQL execution, then is called function *TryLock*, which description is in subchapter 8.2.1. If this lock is according to worker state and transaction successful, then worker changes it states into state *Locked* and waiting for SQL statements to execute. When all requested SQL statements are executed, and all handles to the interface of this worker are released (description in 8.2.2), then worker change states back to *Ready* and again waiting for next actions and checking its connection to the database. Next event by which dispatcher can wake up a worker for work is the event that in its commands list, or a default command list are new commands, which should be processed. When this event occurred, then worker changes state to *Execute commands* and begin executing these commands.

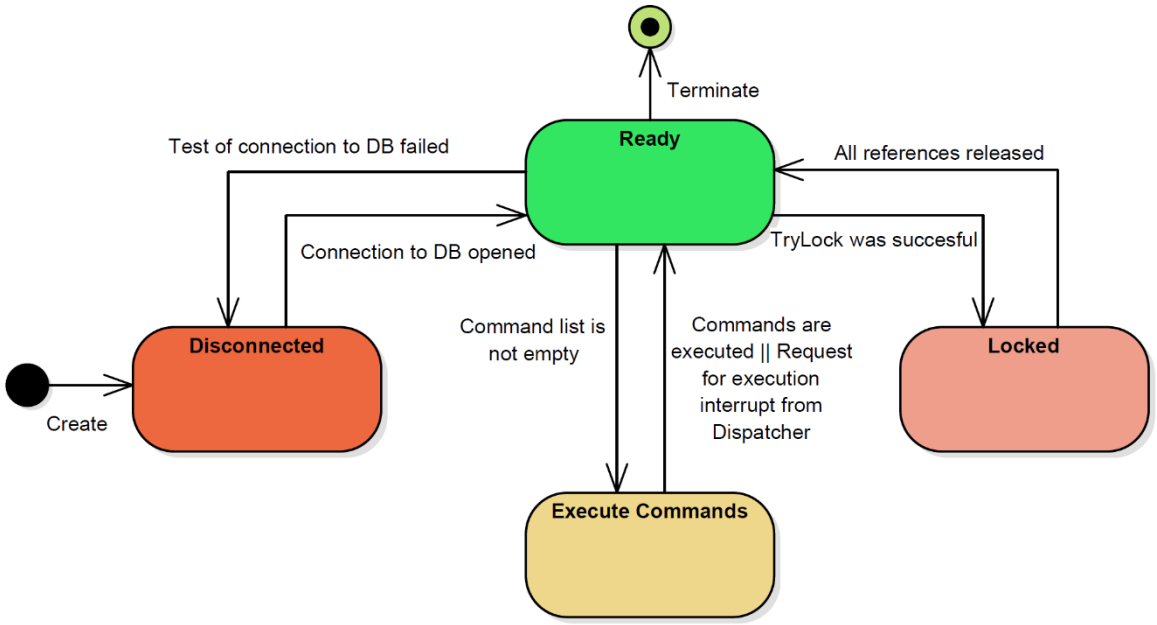

Figure 21 TASWorker state diagram

To change state back to state *Ready* from *Execute commands* are here two possible events. First one is that worker processed all commands and now has nothing to do. Moreover, the second way is that this execution is interrupted by the dispatcher because this worker should be locked for SQL execution. This second case happens mainly when a limited number of workers is available for processing.

8.2. Locking and unlocking worker

As was mentioned in previous chapters, in the library are two possible ways, how workers can execute SQL statements asynchronously. The first way is to add a new command into the commands list, and the worker then executes it. The second way is to lock this worker and gradually assign to it SQL statements for execution, which means that when worker executed one SQL statement, then assign to it new SQL statement is possible. Assign more SQL statements in the buffer of the worker to processing is impossible in this mode. For this approach are here commands lists, which works precisely in this way.

In this subchapter is described the approach with locking and unlocking workers with the gradual entry and executing SQL statements. Here is described function *TryLock,* which serve for locking workers, with its parameters and functionality. Moreover, at the end is described how is used the interface for automatic worker release, which is big positive for the user because he does not have to deal with correct worker unlocking and context releasing because everything is automatic. How it works is described in subchapter 8.2.2.

8.2.1. TryLock – rules for locking

TryLock is the primary function which decides if the worker will be locked or not. This function is a private function of *TASWorker,* and the dispatcher calls it. This function has three input parameters and returns boolean, which characterises if the worker was successfully locked or not. Between input parameters becomes one of three transaction types described in chapter 8.5, one boolean which determines if is possible to change transaction level or not and the last parameter determines if is necessary to interrupt commands execution for thread lock. The dispatcher calls this function with different parameters according to how critical the situation is. The last two parameters are implemented mainly for faster worker locking because when it is necessary to change the transaction level, the transaction must be firstly committed and only after that should be changed transaction level. Also, in interruption of commands execution is necessary to wait until the worker executes the last command and notify that should be safely locked. Is much better to lock the second worker which has same transaction level and is in the *Ready* state, then the first thread which currently processing commands from the list. However, how dispatcher chooses worker for locking is described in the next chapter about dispatcher. For now, it is just important that worker has function *TryLock* with three input parameters used for locking workers to asynchronous SQL statements processing.

The algorithm according to which this function work shows *Figure 22*. Is crucial that whole decisions are performed inside a critical section and the worker cannot change its state in the middle of this algorithm. Firstly, is looked at actual worker state, where it is impossible to lock worker which is currently locked or is in the *Disconnected* state. When a worker is in state *Ready* and has the same transaction level, then nothing interferes with locking this worker and is immediately locked. When

a worker is also in *Ready* state but has different transaction level, then is checked the second parameter with which was this function called and when is enabled to change transaction level then is immediately changed, and the worker is locked otherwise, this function ends without locking. The last option is that the worker is in state *Execute commands*, where also depends on the last parameter. When is enabled to interrupt commands execution, then is interrupted and the worker is locked, otherwise, also this function ends without locking this worker.

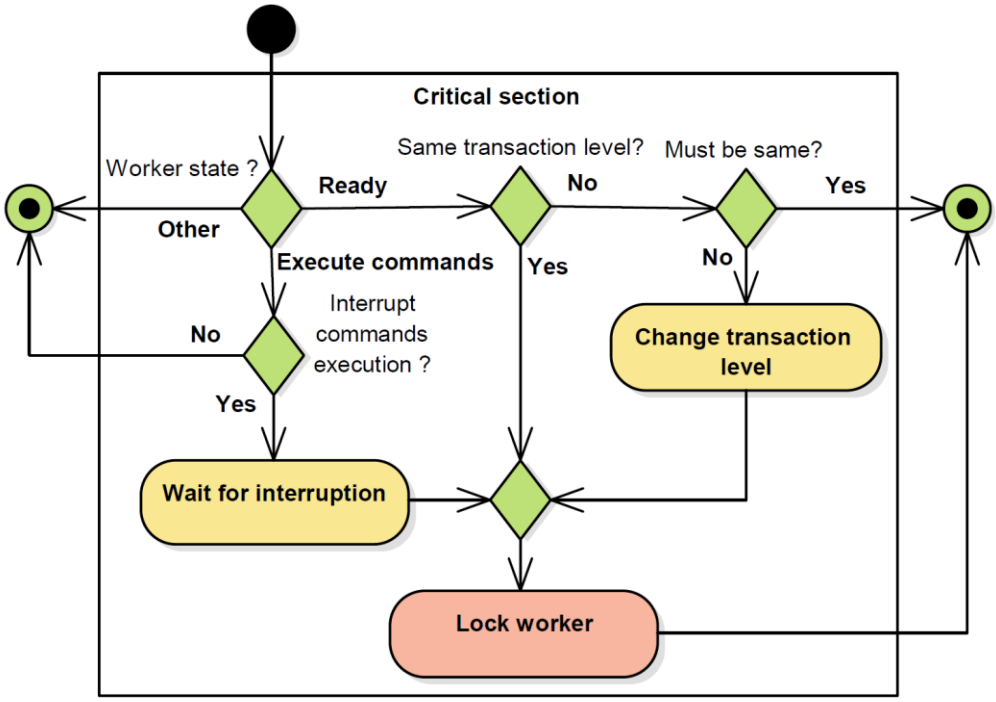

Figure 22 Activity diagram for function TryLock from TASWorker class

8.2.2. Using of interface for worker unlock

After a successful lock of the worker, dispatcher returns to user interface *IASWorker* of this locked worker, by which he can execute SQL statements. One advantage is that user can call only a few functions which are essential for him and the second is that in interface works reference counting, which is used for automatic worker unlock so the user does not have to worry about worker unlocking at all.

For reference counting serves two essential functions which are defined in the primary interface and must be implemented in class which implement this interface. Because the worker is inherited from class *TThread* and this class does not implement any interface, these functions must be implemented in class *TASWorker*. The first function is called *_AddRef* and is called whenever the reference of this interface is assigned into some new variable. When is this reference cleared from a variable, or variable itself disappears, then is called the second important function called *_Release*. These features are used for automatic worker unlocking. The worker has its private integer variable called *FRefCount*, which value is increased whenever function *_AddRef* is called and on the other side is decreased when function *_Release* is called. When the value is decreased back to 0, then it is mean that all references to the interface of this worker were released and unlock of this worker is possible.

Following three figures show how this theory works practically. For this example, was implemented the basic procedure (*Figure 23*), where is assigned interface of the locked worker from the dispatcher to a local variable, then is executed one SQL statement and procedure ends. *Figure 24* is a screen of execution of procedure *GetContext*. It is a dispatcher procedure which returns the interface to the locked worker. Is possible to see that at the end, when is interface assigned to result, is row marked with blue colour, where is called function *_AddRef* in a worker whose interface was passed as a result. Then in *Figure 25*, which shown code from the end of this procedure with an example, is the next important assembler command also marked with blue colour. This command call function *_Release* of worker used for execution and because functions *_AddRef* and *_Release* was called once, then the value of *FRefCount* is back 0, and the worker is successfully unlocked.

```
procedure TMainForm.btnAutoReleaseClick(Sender: TObject);
var w:IASWorker;
begin
  w:=Dispatcher.GetContext(trReadCommitted_ROCommit);
  w.ExecuteSQL('SELECT * FROM Employee WHERE Emp_No = :empno',[114],
    procedure(Result:TelCachedIBSQL)
    begin
      WriteToMemo('SQL was executed, First name is:'+Result.FieldByName('FIRST_NAME').AsString);
    end);
end;
```

Figure 23 Source code for the test with automatic worker unlocking

```
elDatabaseManager.pas.901: Result:=GetWorker(Transaction);
0074D17F 8A55FB          mov dl,[ebp-$05]
0074D182 8B45FC          mov eax,[ebp-$04]
0074D185 E882FCFFFF      call TASDispatcher.GetWorker
0074D18A 8BD0            mov edx,eax
0074D18C 85D2            test edx,edx
0074D18E 7406            jz $0074d196
0074D190 81EA40FFFFFF    sub edx,$ffffff40
0074D196 8B45F4          mov eax,[ebp-$0c]
0074D199 E8820ECCFF      call @IntfCopy
elDatabaseManager.pas.902: end;
0074D19E 8BE5            mov esp,ebp
0074D1A0 5D              pop ebp
0074D1A1 C3              ret
0074D1A2 8BC0            mov eax,eax
```

Figure 24 Assigning reference of worker interface to a local variable of the procedure

When for the same situations will be used some global variable or class variable of the example procedure, then will also be called *_AddRef* after worker locking, but the *_Release* function is not called automatically at the end of the procedure, because the reference to the interface is still held in class or global variable. The worker is still locked until the user not assign *nil* value to this variable, or in case of classes is not destroyed instance of this class.

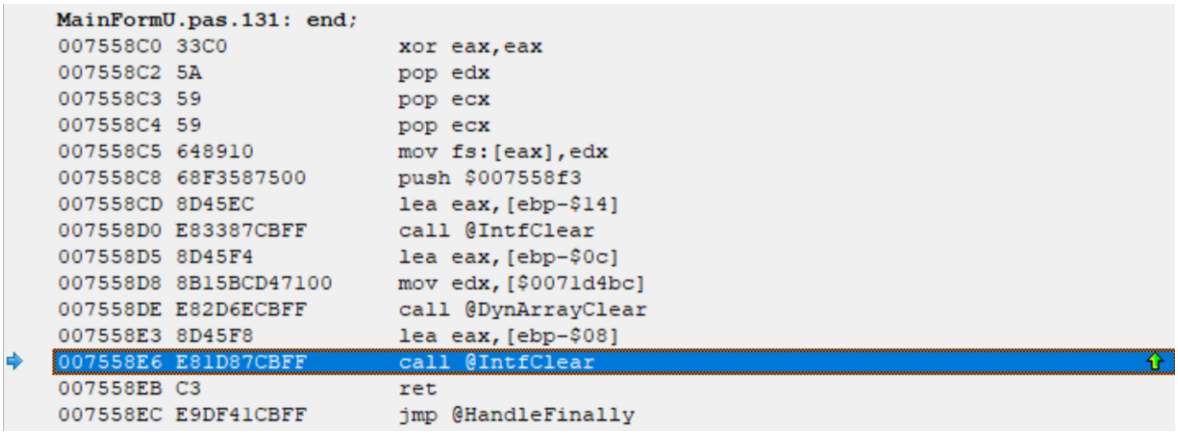

Figure 25 Automatic reference release at the end of this procedure

8.3. Worker interface for the user

As was mentioned in previous explanation how this library and workers works, when the user needs some worker for SQL execution, then he receives from dispatcher interface to the locked worker, by which he can execute SQL statements asynchronously. According to a solution with worker automatic unlocking, there was not found so many functions for the interface. So the interface has only two overloads of procedure *ExecuteSQL*, which the user can use for entering new SQL statements to the locked worker for execution.

The first overload has three input parameters, where the first parameter is a string with SQL statement, the second parameter is an array of variant values which represents parameters for SQL statement, and the last parameter is an anonymous method which is described in the next subchapter. The second overload has only two parameters, where the second one is also an anonymous method, and the first parameter is class inherited from *TASStatement*. Description of this class is in chapter 9.3 because it was developed mainly for testing SQL statements, but basically in each descendant of this class is stored SQL statement and parameters required for SQL executions.

8.3.1. Processing results of SQL execution - Anonymous method

For processing results of SQL statements is used a technique called the anonymous method. This method is a standard procedure which is passed as a parameter with SQL statement and its parameters into database worker for asynchronous processing, and when worker finishes its job, then it calls this anonymous method, where is implemented processing of the result of the execution. Anonymous methods are called from a worker in the *Synchronize* method, which means that they are then processed in the main thread. This approach was chosen because in the anonymous method can be performed for example some actions with graphical objects from GUI, which must be performed in the main thread. Reasons, why is not a correct way to perform actions with GUI from other thread than the main thread, were described in the chapter about multitasking and high-performance application.

To process the result of SQL execution was chosen anonymous procedure with one input parameter which is the variable type of *TelIBSQL*. An instance of class *TelIBSQL* is used for the execution of SQL statements in database worker and then is passed into an anonymous method for result

processing because it has proper procedures for work with this result of the execution of SQL statement. Between the basic procedures belongs procedures called *Next* and *FieldByName*. *Next* procedure performs shift at the next row for processing and by using *FieldByName* is possible to get the value of a field from that row where is now processing located. To find out that the last row for processing was reached is possible to use property *Eof*, which means the end of the file.

Anonymous methods are often written inside calling of *ExecuteSQL* procedure, which means that they are written inside some function or procedure. The question is if the anonymous method can access variables from the method from which is called? The answer to this question is in *Figure 27* and *Figure 28*. For this test was created a procedure with three integer local variables and one string local variable in superior procedure from which is called *ExecuteSQL* function with a created anonymous method. Inside the anonymous method was implemented access to these local variables and was created one local variable inside the anonymous method. Source code for this test is in *Figure 26.*

```
240 procedure TMainForm.btnGetAsyncClick(Sender: TObject);
    var id,id1,id2:Integer; s1:string;
    begin
      id:=116;
      id1:=120;
      id2:=128;
      s1:='Hello';
      IWorker:=DM.AsyncDatabase.GetContext(trReadCommitted_Commit);
      IWorker.ExecuteSQL('SELECT p.Name FROM People p WHERE p.id = :id',[id],
        procedure(result:TelCachedIBSQL)
250     var a:integer;
        begin
          if id = 116 then
            a:=5;
          if id1 = 120 then
            a:=10;
          if id2 = 128 then
            a:=15;
          if s1 = 'Hello' then
            a:=21;
260     end);
    end;
```

Figure 26 Source code to test an anonymous method

In *Figure 27* is displayed assembler code with the operations inside the superior procedure, where are only assigned some values to local variables. The important thing is the base address where these values are stored. Is possible to see that local variables of the superior procedure are stored on base address “ebp-$0c”, which in this case is address “03BE8328”, which is possible to read from “eax” register in *Figure 27*. Then is this base address, stored in register “eax”, used for access to individual variables using relative addressing from this address. For example for integer variable „id1“ is used „eax+$14“, for string variable „s1“ is used „eax+$0c“ and so on.

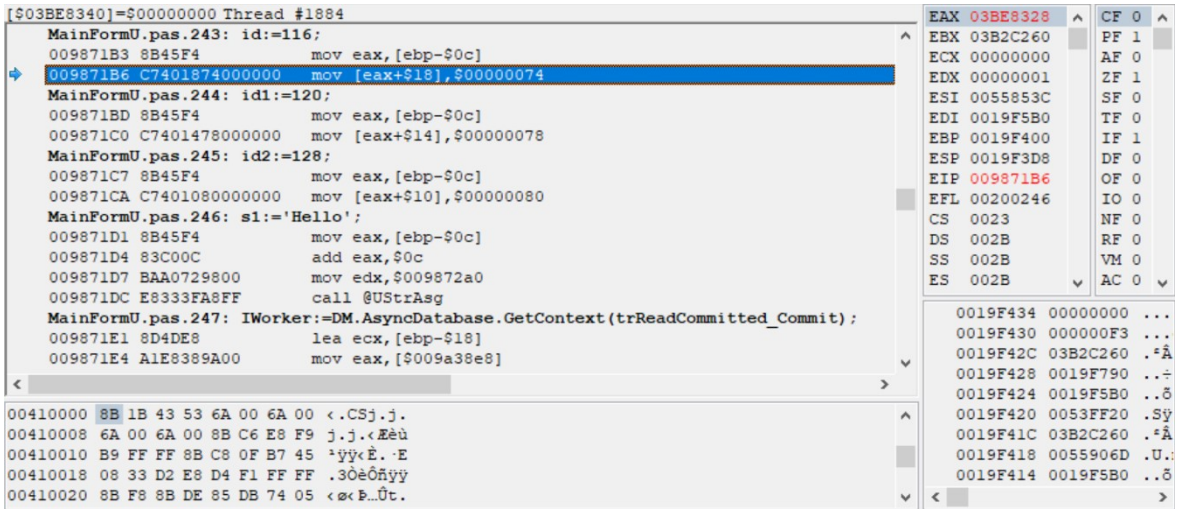

Figure 27 CPU debug information with operations from the superior procedure

In *Figure 28* is then shown assembler code from the anonymous method. In this code are just compared local variables from the superior procedure with the same constant as was there previously assigned and if they are equal, then is assigned some constant into anonymous method local variable "a". From *Figure 28* it is possible to see that the anonymous method has different stack pointers "ebp" and "esp" than the superior procedure. It should mean that access to local variables from the superior procedure is lost, but when it looked well on the values of registers, then is possible to see that base address, which was used for access to local variables in the superior procedure, was restored before calling of anonymous procedure into register „eax". Then is also possible to see from assembler code, that this value is stored to the address „ebp-$04" at instruction „00987101", which is then used as a relative address for access to concrete local variables from the superior procedure as it was inside the superior procedure. At the end it is possible to see that for local variables of an anonymous method is used the different base address „ebp-$08", which means that they are in stack located separately.

The last question around this topic is what happened if the superior procedure is called again although an anonymous method of the previous calling was not processed? Will have an anonymous method from first calling available values of local variables from its first calling, or they will be overridden with second calling, or new calling will have different stacks, and everything will be OK? To find the answer has implemented the procedure with one string local variable, where is stored the time of the calling of superior procedure which is also written into a log file. Then in the anonymous method is read this local variable of the superior procedure with a time of call, which is again written into a log file. For this experiment had to be added 10 seconds sleep into workers execution. Otherwise, it is not possible to perform this test. The result of this experiment is possible to see in *Figure 29*.

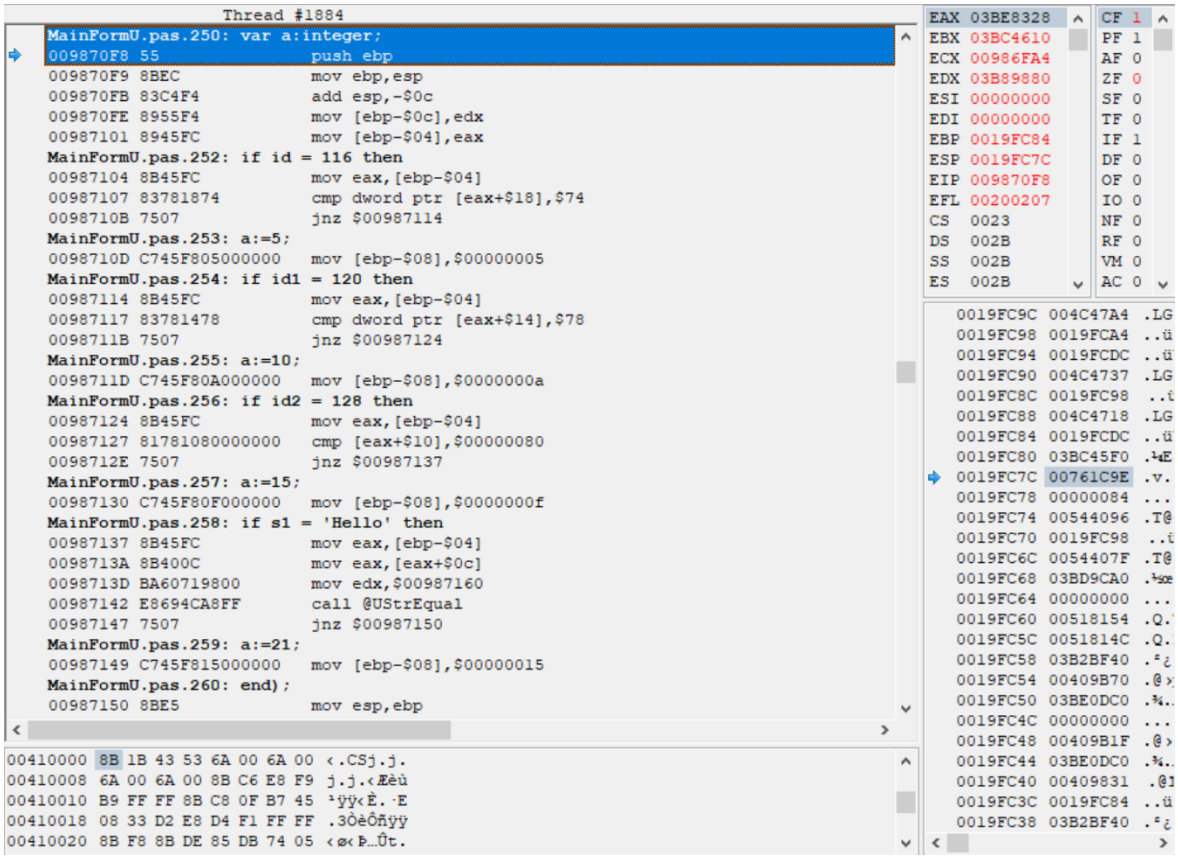

Figure 28 CPU debug information with operations from an anonymous method

16.02.2019 21:05:55	Starting processing asynchronous command
16.02.2019 21:05:58	Starting processing asynchronous command
16.02.2019 21:06:00	Starting processing asynchronous command
16.02.2019 21:06:06	Processing anonymous method of command which started 2/16/2019 9:05:55 PM
16.02.2019 21:06:08	Processing anonymous method of command which started 2/16/2019 9:05:58 PM
16.02.2019 21:06:10	Processing anonymous method of command which started 2/16/2019 9:06:00 PM

Figure 29 Result of experiment with calling the procedure with an anonymous method again though the previous call isn't finished.

Is possible to see that the content of local variables is stored for each call separately, which mean they each call has different stack pointer. So, using of this concept with anonymous methods is save and is possible to call methods with asynchronous SQL statements processing regardless of whether the previous call was completed or not, which is a mainly welcome feature in cases with problems around the connection to the database.

8.4. Execute procedure

This procedure is the heart of the worker according to how works thread in Delphi programming language. After worker creation is immediately started this procedure in the newly created thread, where is performed worker logic in while loop around and around until on this worker is called procedure *Terminate*. After calling of this procedure is set boolean *Terminated* to *true*, which must be checked in while loop for correct termination and worker destroying. How this procedure works and how to look its algorithm is described in subchapter 8.4.1, where the specific function or other essential functions called from this procedure, are described in next subchapters.

8.4.1. Worker running cycle and SQL statements execution

Figure 30 states algorithm according to each worker running. As was mentioned before, the worker runs in this procedure around and around until is set boolean *Terminated* to *true*. At this moment the worker finishes its work, and its destruction is possible. In the first step, a worker checks if it has opened a connection to the database and if not, then immediately open it. In case that connection was not successfully opened is generated exception which must be caught and correctly handled. Otherwise, the worker is aborted. In exception handler of this part of the code, in case that worker is locked and is required to process SQL statement, is this execution immediately cancelled, an exception is raised to the main thread as a result of execution and worker go back to the beginning of this procedure to start open this connection again. In case that worker is not locked is just set the state to *Disconnected* and worker also goes to the start of this procedure to try again open the connection to the database.

When the connection is opened, the worker goes to the next step where test the connection to the database only in case, that is in a *Ready* state. If the connection test fails, then is a connection to the database closed, the state is set to *Disconnected,* and the worker goes back to the beginning of the procedure. If this test is successful or worker is locked, then it goes to next step where is checked how long is opened transaction and in some critical times is performed the appropriate action, but this topic is deeply described in subchapter about transactions (8.5).

After this last step worker comes to the part of the SQL execution. When worker should execute some SQL statement or test registered SQL statements, then worker enters into section bounded with exception handling. In this section is opened transaction in case that is closed or just increased transaction tag which signal that worker continues to execute SQL statements in an opened transaction. After this step are executed SQL tests or SQL statement according to which bit from these two is set to *true*. In case of SQL execution if was assigned some anonymous method for processing SQL result, then is this procedure called in the main thread by *Synchronize* procedure. When everything was executed without exception, then the worker goes further in its algorithm, but in case of an exception, the exception is correctly handled. When an exception occurs, then is immediately cancelled SQL execution and exception is raised to the user as the result of SQL execution. In case that this exception was caused due to interrupted connection to the database, then is this connection closed, a transaction tag is set to 0, because the transaction is automatically committed when the connection is interrupted, and worker goes back to the beginning of this procedure where try to re-establish database connection.

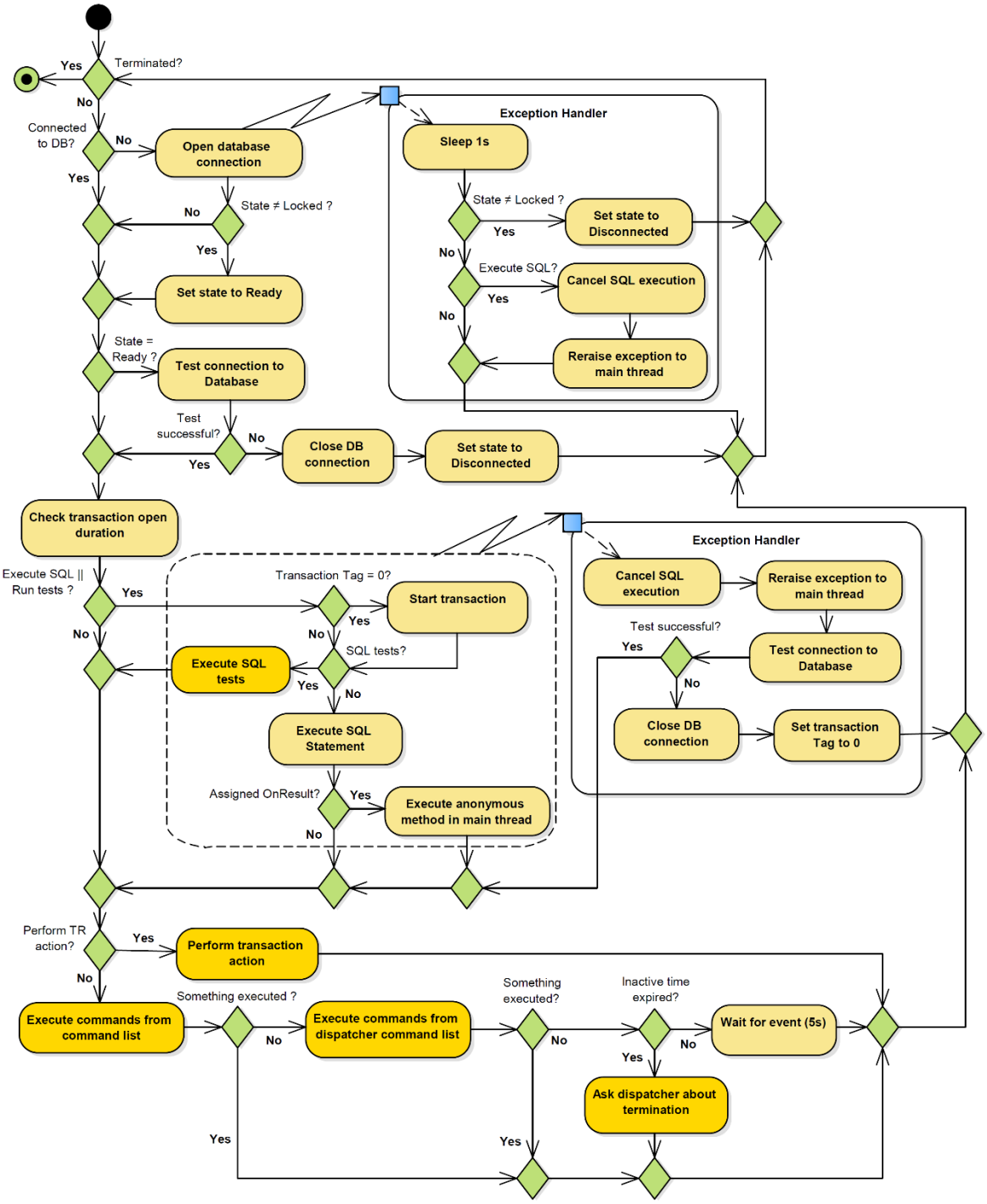

Figure 30 Algorithm of worker execute procedure

In the next steps of this algorithm, a worker could perform transaction action when was released all contexts to this worker and it was unlocked. If this action is not required then worker trying to check command list and if there are some commands, then it executes them and returns to the beginning of the procedure. When a worker has not any own commands to execute in the list, then it looks if are not some command in dispatcher default command list to execution, and if they are there, then worker executes them. If neither in a global command list is any commands for execution, then worker

checks how long is inactive. If this time is more than max allowed time in the setting of the dispatcher and worker is not locked, then is called procedure *WorkerTimeExpired* which belongs to the dispatcher and determines if a worker should be terminated and destroyed or not (9.4). When dispatcher decide that worker should be destroyed, then dispatcher returns true as the result of this procedure and destroy this worker, which set boolean *Terminated* to true. Then the worker goes at the begin of the procedure, leave this procedure *Execute* and is successfully destroyed.

Explanation about this functionality and why is implemented is in the chapter about dispatcher, here is just described how it works from the worker side. If the worker did not even reach the time for destroying, then it goes sleep and is wake up after 5 seconds, or also may be wake up by an event from the dispatcher.

It was the explanation of how works the cycle in which the worker continually runs. How works SQL execution and partially error handling in unexpected situations should be understandable from this moment. In this algorithm was also described when are executed some special methods as the procedure for SQL testing, the function for commands execution, or something about the transaction. However, capture everything into one figure is impossible and these topics describe the next subchapters.

8.4.2. Commands Execution

For the execution of SQL commands serve function called *ExecuteCommands*, which is a private worker function. This function has one input parameter to passing thread save command list for processing. In case of processing worker internal command list is called this function directly, but in case of processing the default command list placed inside dispatcher, is called an auxiliary function called *ExecuteDispatcherCommands*. Here is also called function *ExecuteCommands* but with the dispatcher command list. This calling is wrapped into a critical section of the dispatcher to prevent situations, where more than one worker processing this command list. Inside critical section worker entering by function *TryEnter* which means that in case that dispatcher command list is processed by some worker, then another worker does not wait until the first worker finishes execution of command list but run further in its algorithm and skip this step.

At the beginning of function *ExecuteCommands* worker locks command list to processing, which means that until command list is locked, then add new commands into it from other threads is impossible. The worker also enters a critical section to ensure at 100%, that in this section will not be worker locked for executing SQL statements. Inside these two critical sections is checked if the command list is not empty or worker is not locked. If one of these conditions is true, then the worker immediately leaves the critical section, unlocks commands list and leaves this function. In case that both conditions are false, a worker set its state to *Execute commands*, set the result of this function to *true*, create a copy of thread safe command list for processing and finally leaves the critical section and unlocks command list.

Then in this algorithm is checked the first command from the command list. In case that this command not failed in execution at a previous time or passed the time after which can be performed new attempt to execution, algorithm passes to the part of the code for executing commands in a loop, boarded with exception handling. In opposite case is checked and performed necessary action with

the failed command, which was requested by the user from the diagnostic form. How this error handling works and what user can do is described in the chapter about error handling in diagnostic form.

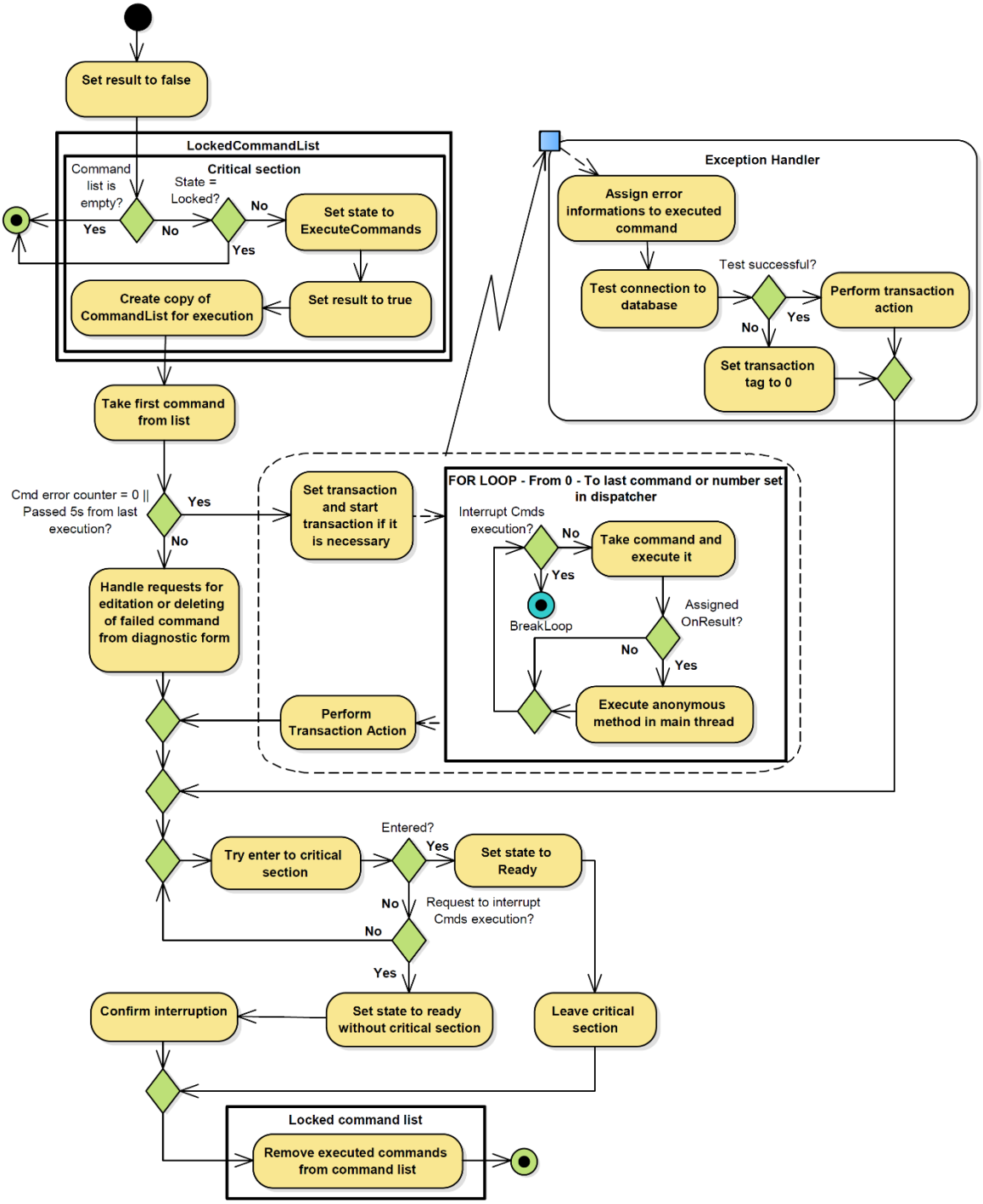

Figure 31 Algorithm of worker Execute commands function

In the area boarded with exception handling worker firstly open transaction if it is necessary and then jump into a loop where execute all commands one by one and in case that they have assigned some

anonymous method, then obviously also call this method in *Synchronize* procedure. This loop is then terminated in case that all commands were executed or the number of maximum allowed execution per transaction was reached or dispatcher request to interrupt this execution because is necessary to lock this worker for the user, or the last possibility is that occurred some exception in execution.

In case that occurred some exception in commands execution, this exception is caught by the exception handler. Because commands must be performed in the same sequence in which was added into command list, then in case of the exception must be this loop broken and next commands cannot be executed until this command is successfully executed. In an exception handler are assigned exception information to failed command for diagnostic form and also is increased its error counter. In case that exception was occurred due to interrupted connection to the database is only set transaction tag to 0, because transaction action is automatically performed on the server side when the connection is lost. On the other side, if the connection is still alive, the worker performs transaction action.

In the next step of this algorithm is changed worker state back to the state *Ready*. The code around changing this state is a little complex due to functionality around interrupt worker commands execution for its locking. If this part of the code was written easier as it is in *Figure 32*, that is here a scarce risk for deadlock, which must be eliminated. How this deadlock situation can happen is described in *Figure 33*.

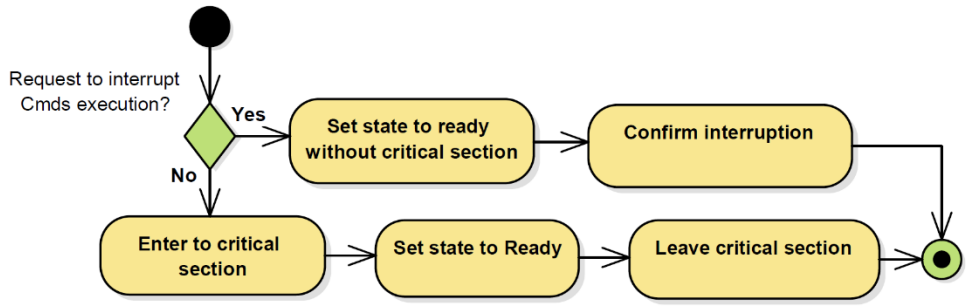

Figure 32 Code in Execute commands which should cause deadlock

In this unpleasant situation main thread, which performs *TryLock* function, entering a critical section where checking the state of the worker, which is now in state *Execute commands*. In the same time, worker which want to set its state to *Ready*, recognise that loop was not broke due to the requirement to interrupt commands processing, so it wants to change state in critical section as it is in every part of the code and enter the critical section. Because the main thread now holds the critical section, then the worker is blocked. At this time main thread notifies the worker that it should interrupt commands execution because it is necessary to lock it for SQL execution. Unfortunately, worker skipped this critical condition and now waiting until the critical section is unlocked, which is impossible because main thread also waiting when gets acknowledge from the worker that commands execution was interrupted, and the worker is ready for the lock. So, deadlock is here.

Due to this risk was implemented a loop in which worker stay until is correctly confirmed interruption of commands execution or changed state to *Ready* in the critical section. In case of interruption is state changed without critical section, because worker known exactly in which part of the code is the

main thread and that is possible to do that safely. If a worker tries to change state in a critical section, then it enters into this section by function *TryEnter*, which not blocked a thread in case that section is locked, and the worker can continue in this loop. This problem should also be solved that in this part is state set to *Ready* in every case without critical section, but from a personal point of view it is not a clean solution, which should cause some other problems in further extensions and new functionalities in this library and for this reason was chosen described solution with loop.

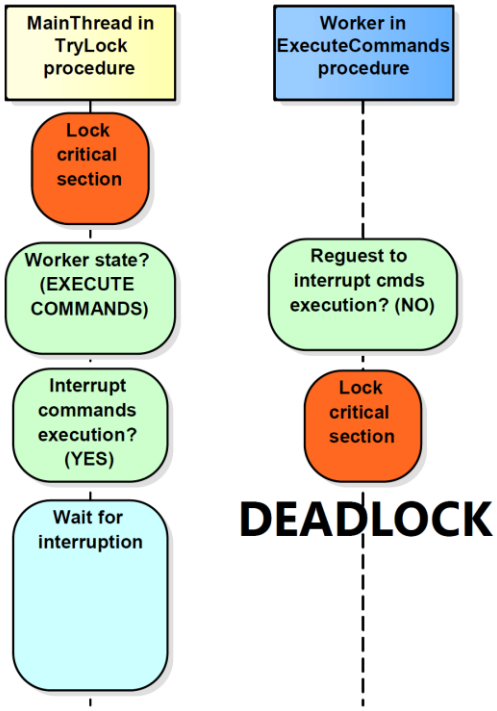

Figure 33 Sequence which should cause deadlock

After a successful setting state to *Ready*, the worker then only deletes all commands which were executed from thread save commands list and ends this function.

8.4.3. Execution of SQL tests

The topic around SQL testing from part of the worker is nothing hard to explain. The worker has prepared from dispatcher list of SQL commands where each command represents one SQL to test with parameters, where all of them are executed consecutively in a loop until all commands are executed, or some test fail, which generate an exception. At this test is each SQL statement prepared, where is checked syntax and when is also required to execute this statement, so it is executed otherwise, worker coming to the next SQL command. In case of an exception is this exception caught, then is added information into exception message about on which statement this exception occurred and finally is this exception re-raised upper, where is handled in the same style as are handled exceptions from SQL executions. In case that is assigned event, which should be called after a successful SQL test, then is this event called in the main thread when the test was successful.

8.5. Transactions solution

The problematic around transactions was thoroughly theoretically explained in chapter 4, where was described what transactions are, why they are used and some models of the transaction. According to this theory was designed solution for transaction handling in this database library. The final solution is not the same as some from the described transaction models but is inspired by them and completely respects ACID rules. As is shown in *Figure 19*, each worker has its connection to the database and one own transaction. In subchapter 8.5.1 is described how the user can work with these transactions and in subchapter 8.5.2 how are handled long live transactions.

8.5.1. Possible transaction levels and actions

In case that user is asking the dispatcher for locking one worker for executing SQL statements, then one of the parameters with which this question is asked is a parameter which determines the type of transaction. In this library are implemented three transactions types which the user can choose:

- Type 1 – *Read committed* + *ReadOnlyCommit*
- Type 2 – *Read committed* + *Commit*
- Type 3 – *Snapshot* + *Commit*

Is possible to see that each type has defined transaction level and transaction action which should be performed after SQL executions. As transaction levels were chosen two the most important and most used levels *Read committed* and *Snapshot*. As transaction actions are common *Commit* and *ReadOnlyCommit*, which signs that was executed only statements which read data from database and transaction should stay open for next operations. Just for an explanation, *ReadOnlyCommit* was implemented in the database library of ElektLabs company, which is used in this thesis. For this feature is used counter called *Tag*, where for true transaction start is started the transaction and *Tag* is increased to 1. For true transaction *Commit* is transaction committed, and *Tag* is decreased back to 0. When is required to perform only *ReadOnlyCommit*, then *Tag* is decreased to 0 without any true transaction *Commit*. Otherwise in case that was called transaction start and transaction is still opened, then is also only increased *Tag* without true transaction start. It is the reason why is in some cases in algorithms only operated with transaction *Tag*.

Procedure *SetTransaction* has implemented for transaction settings which single parameter is the type of transaction to set. In this procedure is checked, if is necessary to change transaction level and if yes, then in case of the opened transaction is transaction committed and transaction level is changed and in case of the committed transaction is just changed transaction level. Everything is performed inside the critical section. The algorithm shows *Figure 34*.

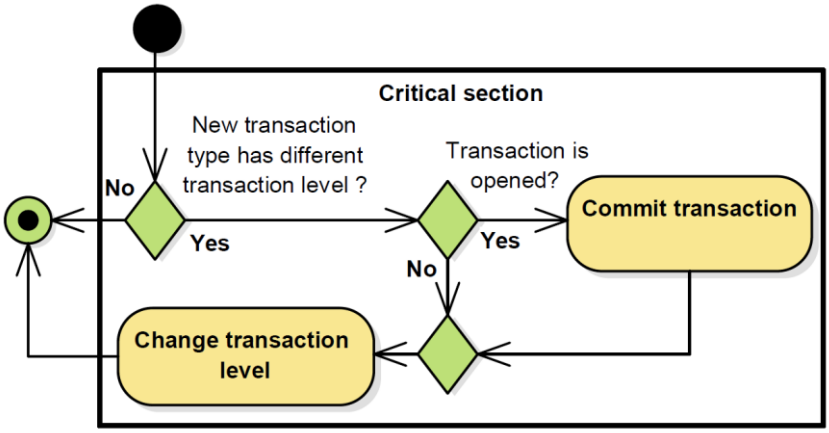

Figure 34 Algorithm of worker procedure SetTransaction

8.5.2. Automatic commit of long live transactions

ReadOnlyCommit is excellent feature how is possible to use one transaction for multiple operations without committing and starting a new transaction, which is very time consuming so with this feature is possible to achieve better performance of database library. On the other side, a transaction which is open too long also is not very beneficial for database server or applications. For this reason, is in ElektLabs library implemented function *CheckOpenDurations*, which receives two input parameters. The first parameter represents time which when is greater than the duration of the transaction, then is planned that the next planned *Commit* will be at 100% true *Commit* no matter if it should be real *Commit* or *ReadOnlyCommit*, the transaction will be committed. The second parameter then defines time which when is higher then the duration of the transaction then is transaction immediately committed. When is function *CheckOpenDuration* called is shown in *Figure 30* with the algorithm of worker procedure *Execute*.

8.6. Silent reconnection

This functionality was implemented for cases that connection to the database is interrupted. In that case, automatically re-establish this connection without any user interference is necessary. The solution of automatic silent reconnection is implemented inside worker *Execute* procedure, so is visible in *Figure 30*. To find out connection interruption soon, worker tests if the connection is still alive in 5 seconds periods when is self-wake up from sleeping and waiting for an event because it has nothing to do. When this test failed, then is immediate connection closed, the state of the worker is set to *Disconnected* to prevent that will be locked for SQL execution and worker start immediately re-establishing this connection.

In case that is not caught this issue by part with testing connection and the worker is locked for SQL execution, then this issue appears in part with execution SQL statements. In this case is thrown an exception from database component, which is correctly caught in exception handler and exception is raised to the user as the result of SQL execution. Then is in exception handler tested if an exception was caused due to connection to the database and if it is true, then is connection immediately closed, and worker returns to the beginning of the procedure, where starts to re-establish database

connection. In case that connection is lost at testing SQL statements, then is expected the same behaviour as in executing SQL statements.

The last possible place where this problem may appear is in case of executing SQL commands. In that case, the exception is assigned to the currently processed command, which was not correctly executed, and execution of SQL commands is cancelled. Then the worker goes back to the start at the procedure *Execute*, where the connection to the database is tested. The test then fails, and the worker tries to re-establish this connection.

For this problematic around interrupting the connection to the database and silent reconnection were analysed lot parts of the code, so when the connection to the database is lost, that in each time worker should re-establish it without any problems and user interference.

8.7. Handling exceptions from SQL execution

Inside the library must also be correctly handled exceptions which occur during the execution of SQL statements. How are handled exceptions which are related to the database connection was described in the previous subchapter but is also possible that is thrown some exception due to the wrongly written SQL statement, database trigger, transaction conflict and so on. As was mentioned in the description of how worker works and how its functions and procedures works, here are two possible ways how are SQL statements executed and in the description of its execution was also outlined how are handled the exceptions, but for clarity here is a summary.

The first possibility of how SQL statements are executed is by locking worker with assigning SQL statement for execution through the interface. When occurs some exception in this case then is user notified about it, and he gets this exception as the result of SQL execution, so this SQL will not be executed again. There are two ways how is user notified that SQL execution was not successful. The first way is that he implements exception handler which is called in *Synchronize* procedure in case of an exception. This exception handler is the procedure with two parameters, where the first is the sender and second is an exception. This exception handler is implemented for the dispatcher, so it is common for all workers. If the user does not implement exception handler, then is only showed a message box with exception message and he must confirm it otherwise, his application is stopped.

The second possibility is that exception occurs due to processing commands from the command list. In this case, is no exception raised to the user but is attached to the executed commands and is showed in diagnostic form. Then is this command executed again and again until it is not successfully processed. The user can modify this command or delete it from diagnostic form, which is shown in the chapter about diagnostic form. Other commands, which are in the list behind this problematic command, are not executed until the problematic command is successfully executed, because the sequence of commands must be observed.

9. Dispatcher

The dispatcher is the main component to which user accesses when he needs to execute some SQL statements, where there are two possibilities of how the user can execute these statements. The first option is that he asks the dispatcher to lock some worker for execution and after locking he can forward SQL statements for execution to this worker, which describes subchapter 9.1. The second option is that the user adds a command to command list via the dispatcher, which is then executed by the worker, where this option describes subchapter 9.2. In the next subchapters of this topics is then described how is solved testing of SQL statements, how are destroyed workers which are inactive for a long time, or which possibilities have a user in case of library configuration.

9.1. Locking workers for SQL execution

Locking workers and forwarding SQL statements to these workers for execution is the first access to how a user can execute SQL statements asynchronously. The interaction between the user, dispatcher, and workers shows *Figure 35*. When the user needs to lock some worker for SQL execution, then he calls the dispatcher function called *GetContext*, which has one input parameter. This parameter determines transaction type (8.5.1) in which is SQL statements executed. In function *GetContext* dispatcher looking for the right worker, which will be locked for the user according to these steps:

1. Find existing worker in state *Ready* and with the same transaction level
2. Find existing worker in state *Ready*, no matter its transaction level
3. Create a new worker if it is possible
4. Find existing worker in state *Execute commands* and force interrupting of this execution for locking
5. Thrown exception that all workers are busy at this time

For this searching serve worker function *TryLock* (8.2.1), which dispatcher call with different parameters according to in which step is now located. Whole steps are performed inside a locked list of workers to prevent situations, that worker is at the same time terminated (9.4), which would cause big problems. In the example is presumed that some worker exists and is already in state *Ready*. Then dispatcher successfully locks this worker and send to the user interface (8.3.) of this locked worker. With this interface is user able to pass SQL statements to the worker for execution. After entering a new SQL statement, the worker immediately starts to execute this statement asynchronously, which mean that the main user thread is not blocked, and it can perform some other work. After successful execution then worker calls an anonymous method (8.3.1), which user assigned to this SQL statement, where the user can process the result of execution, or enter new SQL statement to the worker for execution. When all SQL statements are executed, and user clear are held references to the interface of the worker, then is worker automatically unlocked (8.2.2). In case that some exception occurs during the execution of SQL statement is user immediately informed about it (8.7), and the anonymous method is not called.

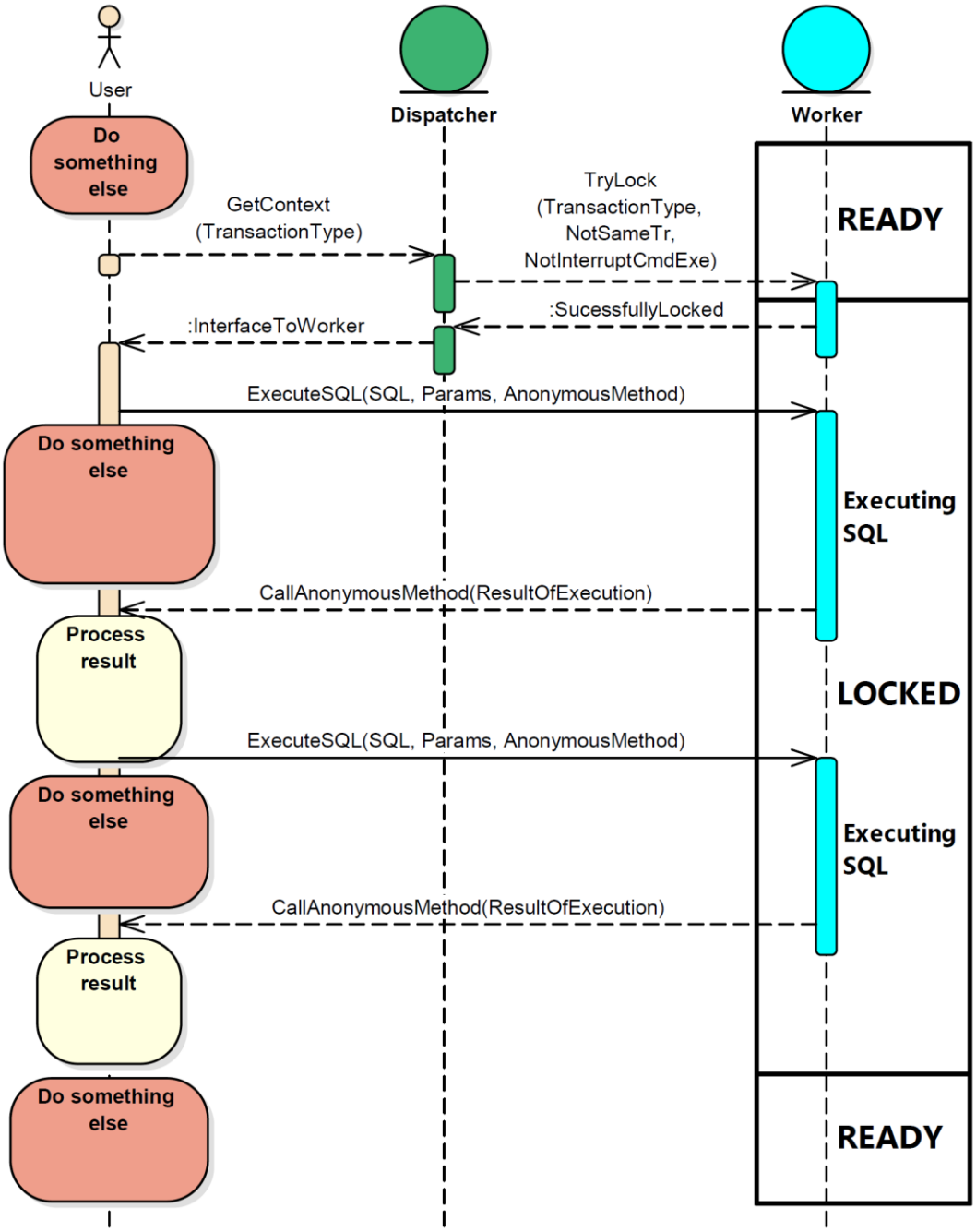

Figure 35 Executing SQL statements asynchronously using worker locking

It was a short example of how can look one relation in which a user can execute SQL statements asynchronously with locking workers. In *Figure 36* is the next example, which shows how workers can change its states, how dispatcher can lock them for SQL execution and when they execute SQL commands, which is the topic of next subchapter. In this example is assumed, that in the configuration are set 2 maximum database connection, 1 minimum database connection and 30s for the maximum allowed worker idle time. What these parameters mean is stated in chapter 9.5 about library configuration.

At the beginning of this example is created one worker for the execution of SQL commands, so worker immediately after creation establish a connection to the database and start executing commands. In time 5s user ask the dispatcher for locking one worker for SQL execution. In that case, dispatcher skips step 1 and 2 in the *GetContext* algorithm because currently exist only 1 worker which is not in state *Ready*. So, the dispatcher creates a new worker and immediately lock it for SQL statements execution. Then in time 10s user ask again for locking of a new worker. In that case, must dispatcher also skip step 1 and step 2, because no workers are in state *Ready*. The third step is also skipped because the maximum possible workers were created. So in this case, must be interrupted execution of SQL commands. For this action is notified worker 1, that is necessary to stop the execution of SQL commands, worker change state to *Ready* and then is immediately locked. When the user finishes his SQL execution with worker 1, then is automatically worker unlocked which means that worker changes its state back to *Ready*. Then worker 1 immediately notice that has some unexecuted commands in the command list and starts executing the remaining commands. The last interesting thing at this diagram is automatic termination of the inactive worker which is performed in time 80s, because the worker was inactive for 30s and in this time are active 2 workers, which means that 1 should be terminated that also happened. How the algorithm for this termination works describes subchapter 9.4.

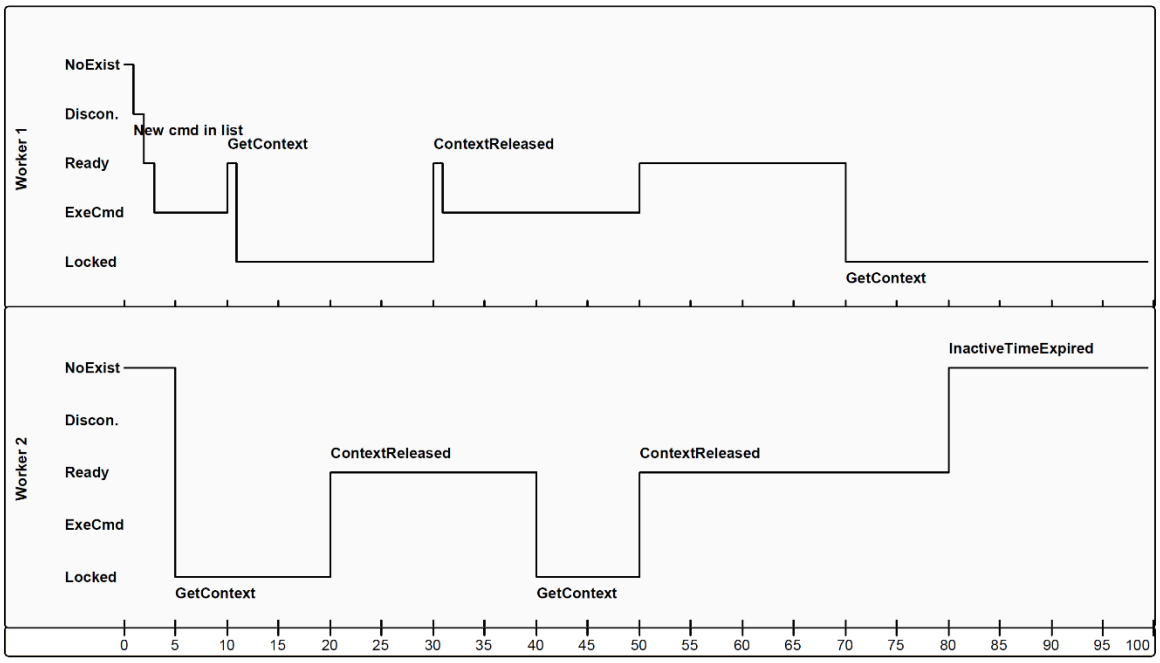

Figure 36 Example of workers timeline

9.2. Adding commands to commands lists for execution

For entering commands to lists serve two overloads of dispatcher procedure *AddCommand*. Both overloads have as the first parameter ID of the list, where is required to add a new command. ID 0 means that is required to add a new command to the default list, which should be executed by a random worker, which at this moment have nothing to do. Then the user can pass as the ID of command list any number from 1 to amount of maximum allowed database connection, where this

command is added into command list owned by the worker with the same ID. The first overload also has other typical input parameters as SQL string, a variant array with SQL parameters and anonymous method, which should be assigned. The second overload has instead SQL and parameters one parameter, which is class inherited from *TASStatement*. This class is used for SQL testing, so is well described in next subchapter, but basically inside this class is also implemented SQL statement and parameters. The second overload as the last parameter also has an anonymous method as the previous one.

In both overloads is new command constructed from these parameters. These commands are the type of *PASCommand*, which is a pointer to record *TASCommand*. After this construction is then called private dispatcher procedure called *AddCommandToList*, where this command is added to the right command list and workers for their processing are waked up or created. This procedure has two input parameters, where first is ID of command list and second is a pointer to the new command. How works this procedure shows *Figure 37*. For whole procedure is locked thread safe list with created workers for preventing the situation that worker is destroyed at the same time when this procedure assigns a command to its list, which causes that this command will never be executed.

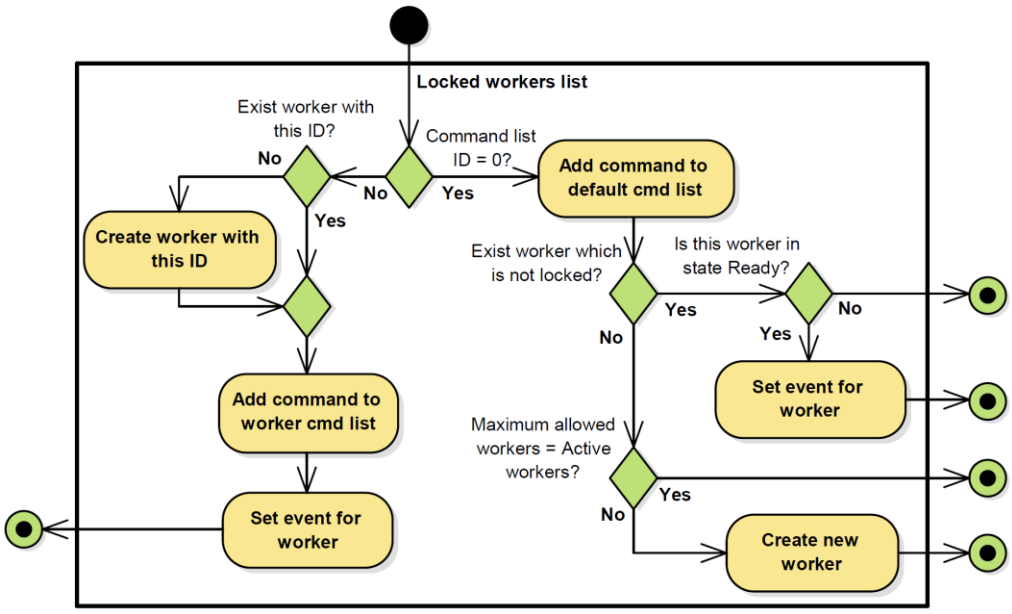

Figure 37 Creation of workers for commands execution

Algorithm branches into two parts where the first condition of this branching is the ID of the command list, where the command should be added. When adding a command into worker list is required, then is first checked if this worker exists. In case that worker does not exist then the worker is immediately created. Then is added a command into worker command list and the worker is notified about it with the event, so it starts executing it. The second option is that the user wants to add this command into the default command list. In that case, is command added to command list and then is checked workers situation for the fastest execution of this command. If exist some workers who are not locked at this time, that in case that worker executes commands dispatcher leaves this procedure because it counts that worker executes this command immediately after actual execution. In case that worker is in the state *Ready*, dispatcher notifies the worker that there is a

command that is necessary to execute, the worker is waked up from sleeping and start doing its job. When no worker is available at this moment, then is created a new worker, if it is possible.

It was summary how a user could add new SQL commands for execution with explanation how are these commands added to commands list and how are workers notified to start executing these commands. How then the worker executes these commands from prepared command lists is explained in chapter 8.4.2.

9.3. Registering SQL statements for testing

For this topic, was an effort to design the best concept, which is very easy to use for the user and he can use the same thing for testing SQL statements, assigning SQL statements to workers for execution, or add it to commands list. For the solution to this problem was developed class *TASStatement*, which has two virtual class functions. Functions are class functions because to use this concept is not necessary to create instances of these classes with SQL statements, so work just with the class itself is possible. The first class function is called *SQLStatement*, where this function returns a string with SQL statement, so a user in a descendant of this class override this function where he writes his SQL statement. The second class function is called *Params*, which returns an array of variant elements. So, in descendant class user can add some class variables which have the meaning of SQL parameters and in this function then compose an array of these variables. For usage in SQL execution then he only needs to fill up parameter variables and pass his class, inherited from *TASStatement*, into prepared procedures for execution.

When the user wants to test these statements, then their registration is necessary. For this is implemented a dispatcher procedure called *RegisterSQLTest*, which has two input parameters, where first is user class with SQL statement and with second parameter user decide if this SQL should be only prepared or also executed. When the user wants to test some SQL statement, then he only fills up parameter variables with some test values and then calls procedure *RegisterSQLTest* with his class with SQL statement. Inside procedure *RegisterSQLTest* is implemented nothing complicated, only is composed SQL command from this class, which is added into command list with SQL statements for testing.

When the user wants to execute all registered test, then he calls dispatcher procedure *RunRegisteredTests*, where dispatcher immediately run these tests in the available worker or create a new worker for this test. Procedure *RunRegisteredTests* has one input parameter with the anonymous procedure, which is called in case that test is executed without fault. How worker performing this SQL test and executing SQL statements from the prepared list describes chapter 8.4.3.

9.4. Destruction of inactive workers

In this library was also implemented algorithm according to is possible to terminate workers which are inactive for a longer time then is set in dispatcher configuration. In the description of the worker running cycle (8.4.1) was outlined when worker calls procedure *WorkerTimeExpired,* which is dispatcher procedure where is decided if a worker should be terminated or not. The algorithm of this procedure shows *Figure 38*. The entire decision is performed when is locked list with workers, which

serve as a critical section, because from this lock is impossible to lock this worker for SQL execution or add some new commands inside it commands list. Firstly, is in this procedure checked how many workers are active. In case that was reached the amount of minimal active connection, then the worker is not terminated. After it is with absolute certainty checked if the worker is not locked for the execution of SQL statements and command lists are empty. If everything is right, then is worker terminated, destroyed and removed from the list of active workers.

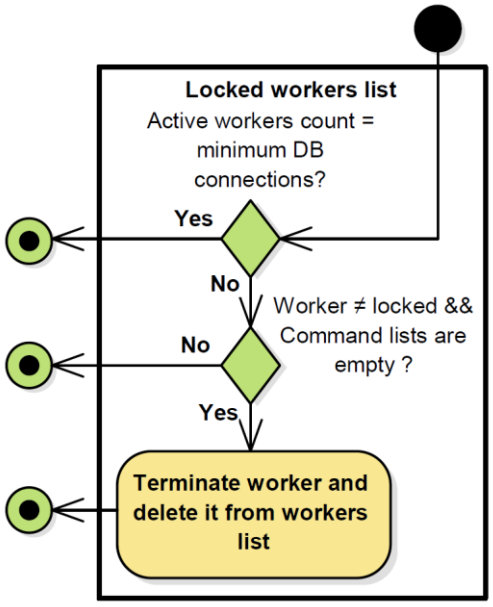

Figure 38 Algorithm for automatic worker termination

9.5. Library configuration options for the user

In *Table 3* are stated all configuration parameters with their description which user of this library must set before he launches his program and starts using this library. The first three parameters must be set otherwise is impossible to connect to the database with the worker. Parameter *Charset* defines charset of the database and is also used for the connection. Following two parameters serve to determine how many workers could be created and on the other side how many workers could be terminated in case that they are inactive for a long time. Parameter *TerminateTime* states how long could be worker inactive before it is terminated. Following two parameters serve for *Commit* of the long-opened transaction, where this functionality describe chapter 8.5.2. Parameter *CommandsPerTransaction* determines maximum commands which should be performed in a row in one transaction. The last parameter called *LoggingEnabled* only disables and enables logging into file and log memo.

Table 3 Library configuration parameters

Parameter name	Description
DatabaseName	Classic Firebird connection path to the database
UserName	The user name for logging to the database
Password	Password for logging to the database
Charset	Charset user for connection
MinConnections	Minimum allowed connections to the database (9.4)
MaxConnections	Maximum allowed connections to the database (9.1)
TerminateTime	When the worker is inactive longer than this time, then is terminated (9.4) [s]
PlanTrueCommit	When is transaction opened longer than this time, then next commit will be true commit in each case (8.5) [s]
ForceTrueCommit	When is transaction opened longer than this time, then is immediately committed (8.5) [s]
CommandsPerTr.	Determine maximum allowed executed commands in one transaction (8.4.2)
LoggingEnabled	Determine if is enabled or disabled logging of debugging information

10. Monitoring and diagnostic GUI

One part of the new database library is also diagnostic form, where is possible to find much information about the configuration of database library, workers states, some statistic data about execution, or errors which occur meanwhile commands execution. In chapter 10.1 is described diagnostic screens with information what they provide. In chapter 10.2 is then described how to handle errors from commands execution when an application is running.

10.1. Diagnostic screens

The diagnostic form with the first screen, which is visible after the first open of diagnostic form, is possible to see in *Figure 39*. This screen shows some important information about the database as a connection path for known where workers are connected, user name with which are workers connected to the database, count of active workers, count of commands from default command list waiting for execution, or how many commands and SQL statements workers executed from the time of application start.

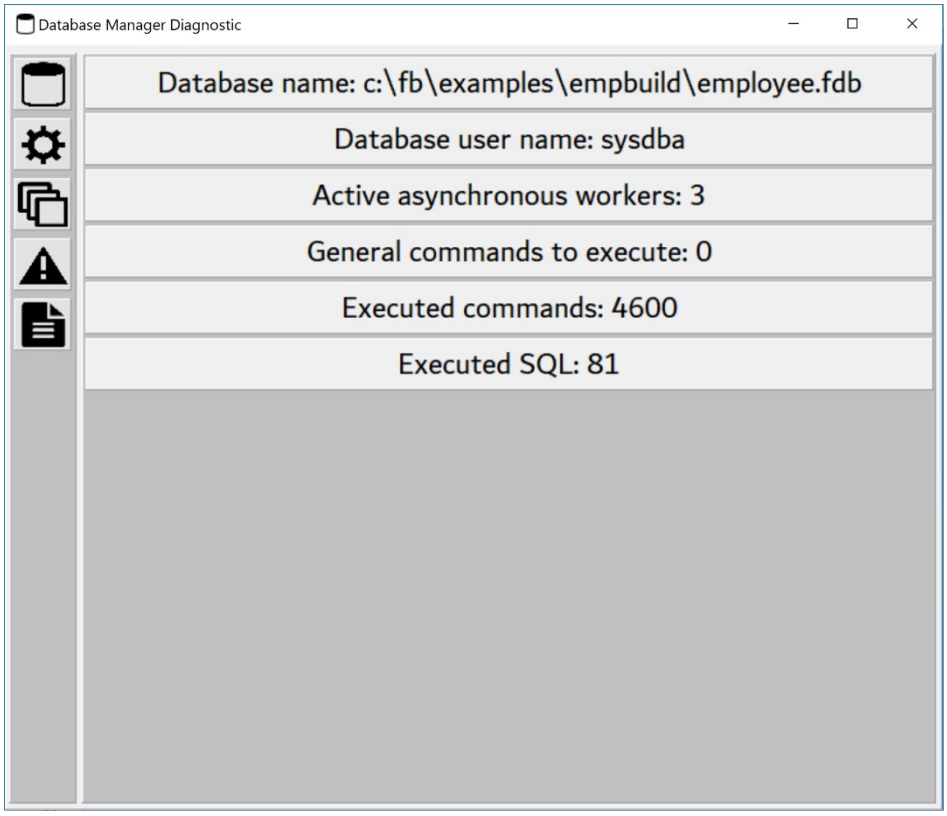

Figure 39 First screen in Diagnostic - Dispatcher information and execution counters

In the second screen (*Figure 40*) is possible to see set values of dispatcher configuration parameters described in chapter 9.5 and on the bottom of this screen is possible to switch on/off writing log information into log file and log memo, which is visible on the last screen of this Diagnostic form (*Figure 43*).

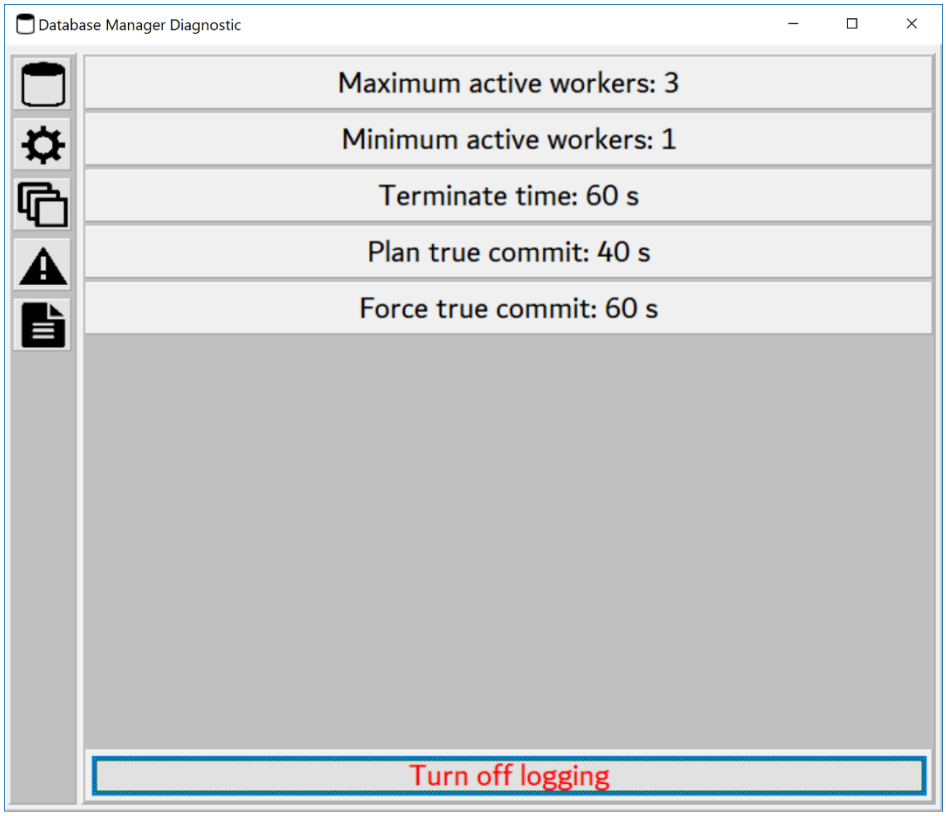

Figure 40 Second screen in Diagnostic - Dispatcher configuration

In the next screen (*Figure 41*) is shown the diagnostic screen with all running workers. It is possible to see that each worker frame provides this information:

- The ID of worker and thread
- Connection to database is established
- The worker is in state *Ready*
- The worker is in state *Locked*
- The worker is in state *Execute commands*
- Number of successfully executed SQL statements
- Number of successfully executed SQL commands
- Number of commands waiting in worker command list for execution
- Elapsed time from last execution of SQL statement or command.

Many details about workers, which execute SQL statements and commands, is possible to know with this information. The configuration of database dispatcher can be then changed for better performance based on this information.

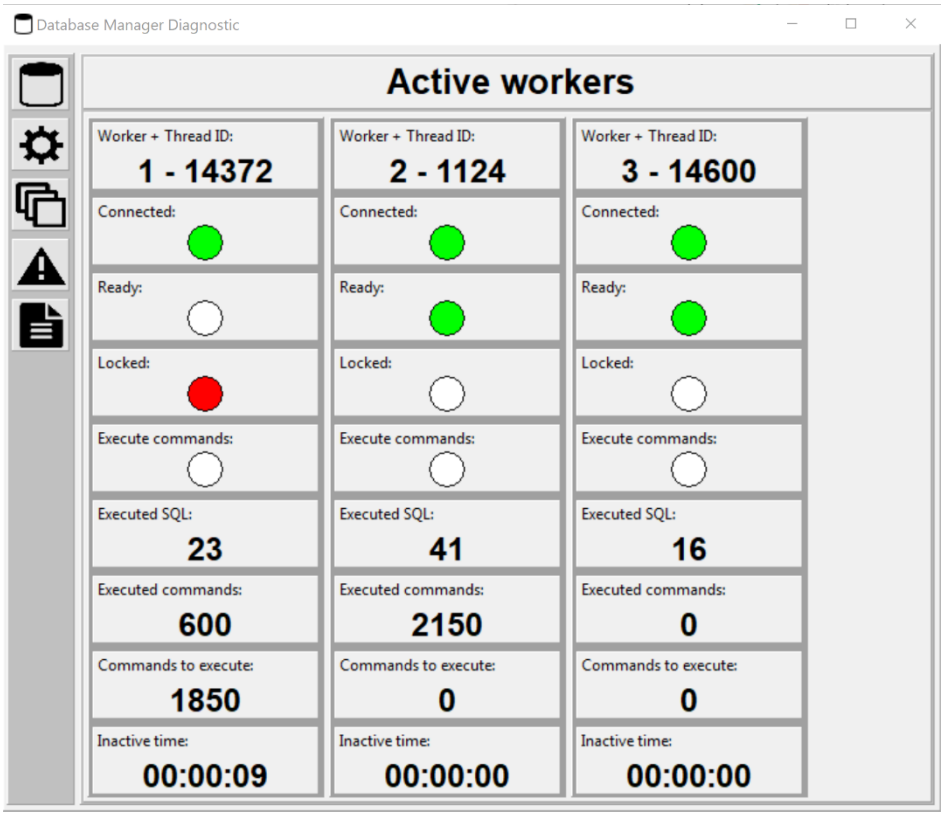

Figure 41 Third screen in Diagnostic – Active workers

Next screen in diagnostic form serves for error handling of SQL commands execution (*Figure 42*). In this screen are visible all occurred errors, where is possible to see information about:

- Failed SQL statement
- Used parameters for execution
- Description of error from the exception message
- Count of failures
- Time of the last failure
- Command list ID, from which command originates

Handle this error and repair what is wrong is then possible with this information, where how it is possible describes the next subchapter.

The last screen of the diagnostic form showing a memo with logged debug information about running workers (*Figure 43*). As was mentioned before, this logging is possible to switch on/off in the setting screen.

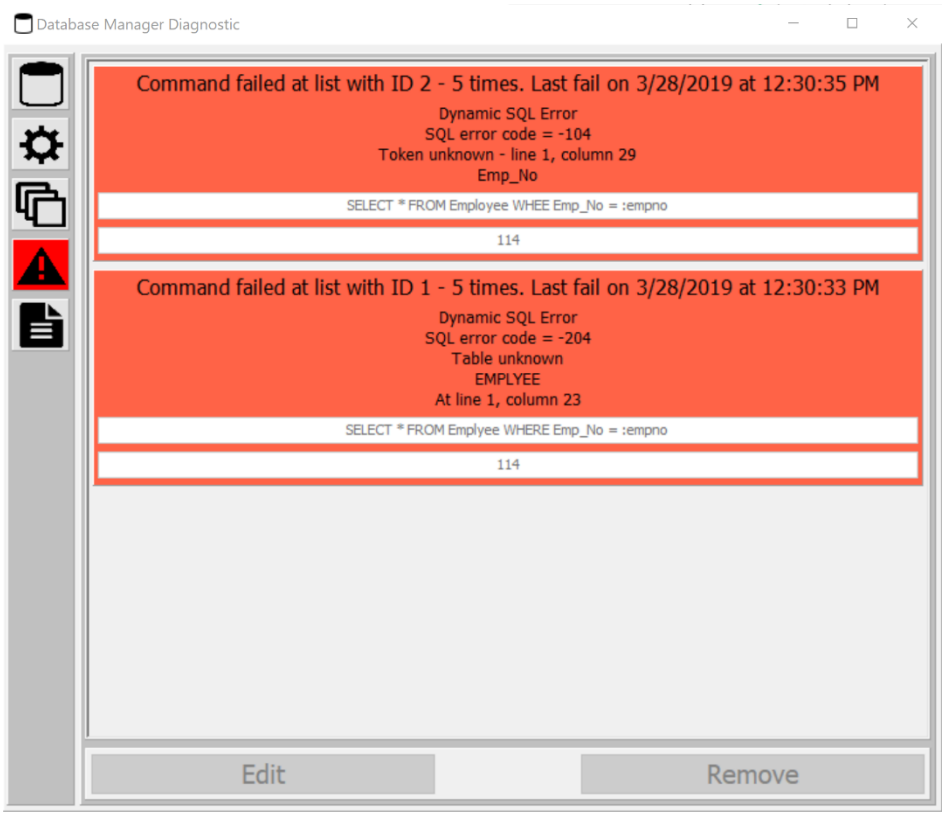

Figure 42 Fourth screen in Diagnostic - Failed SQL commands

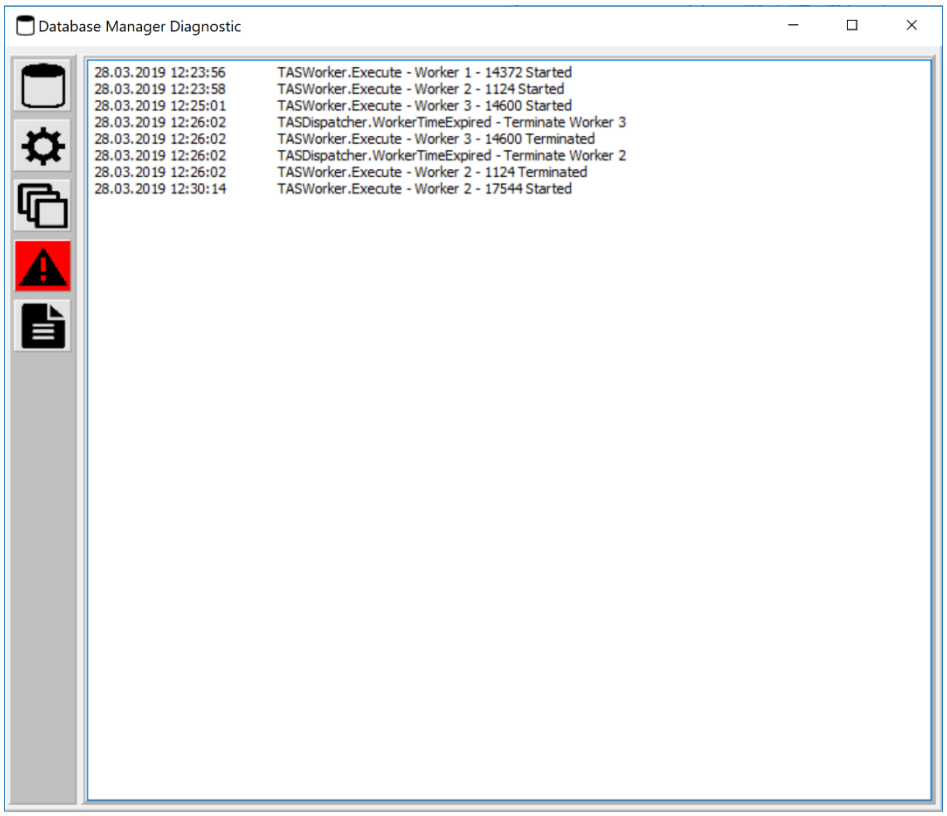

Figure 43 Fifth screen in Diagnostic – Log memo

10.2. Editing failed SQL commands in running application

As was mentioned in the previous subchapter, in diagnostic form exist one screen where it is possible to see all SQL commands, which are not currently possible to execute due to some issue (*Figure 42*). After a click on one error get this error frame red colour and are enabled two buttons with possible actions with this command, which is possible to see in *Figure 44.*

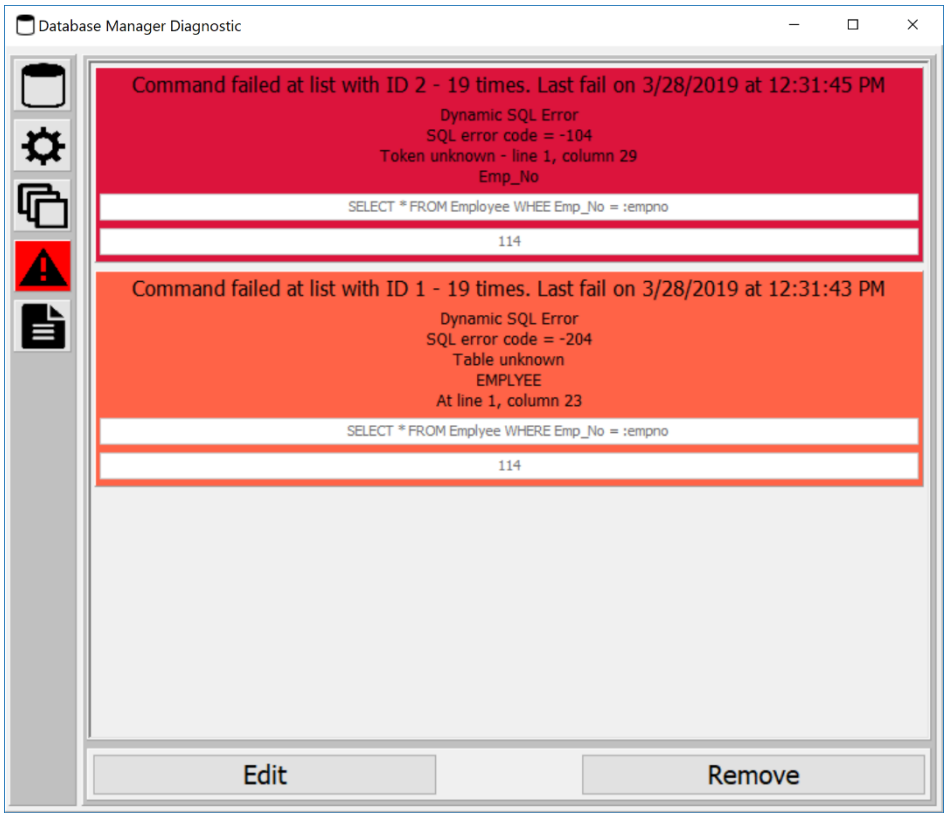

Figure 44 Handling failed commands - choosing the failed command

The first possible action is to edit the SQL statement or parameters as it shows *Figure 45*. In SQL statement should be the same mistakes as it shows the example on the figure, concretely poorly written word *where* causes first error and the second statement works with a table which does not exist in the database. The possibility to edit SQL statements is here mainly by these reasons. However, from practice is possible that also parameters, with which was SQL statement called, should be wrong. On one side is a possibility to paste the wrong parameter type. For example, pass some string into an integer field. On the other side, when types are correct, the statement should fail at database trigger when being passed unexpected values. In this new library are two possibilities of how to fix this problem. First one is to edit parameters, or SQL statement or both with diagnostic form and this command finish with success. The second option is to use the second button which removes this command from the command list and then it is not executed.

Now is in the company used the basic concept of one thread in which context is performed a narrow range of specified SQL commands and which will be replaced by this new database library. The solution to the new library should be very comfortable for programmers. In this nowadays used

thread, when occurs some problem in the execution of one SQL command and programmer wants to fix it, then he has just one possibility. It is that he must switch off application, then found in an XML file with stored commands this problematic SQL command, edit it and then start the application again. If the problem is still active, or appear some another problem, the programmer must perform this action still around, until everything is fine. With using new database library is possible to repair these failed commands from the diagnostic form without an application restart.

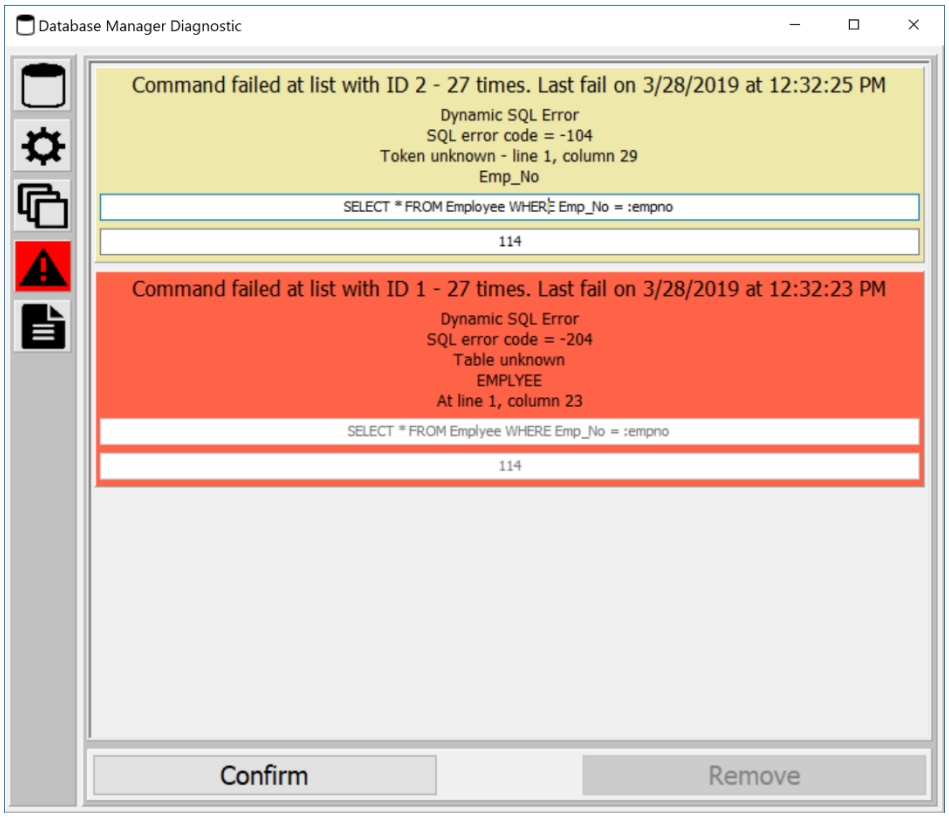

Figure 45 Handling failed commands - editing SQL statement and parameters

11. Library testing

During the implementation of the database library and after that, was library tested. In the first subchapter is described how was implemented automatic unit tests for the main functionality of this library and in the second subchapter is explained how this library was also tested for some basic functionality in a production environment

11.1. Automatic unit tests

During implementation was implemented unit tests for automatic testing of the main functionality of the database library. For test implementation was used standard Delphi test framework, which is also used for unit tests in other projects. When the use of this framework for tests implementation is required, so must be created a new class which is inherited from *TTestClass*. When this new class is registered into *TestFramework* in the initialisation part, then after the test GUI run this class appears between other test classes. These tests are also possible to run in the console so that they are used for automated tests on the company build server.

Figure 46 shows all tests which were implemented after successful test execution. The first test checks the functionality around interruption worker commands execution for its locking. In this test is passed several commands to worker list for execution, where is tested if the worker was successfully created and started executing these commands. After a while is called procedure *GetContext* for the test if the worker stops commands execution and is successfully locked. Then is released handle to worker interface and is checked if the worker was automatically unlocked and starts again executing SQL commands. In next test, which is called *CheckMinMaxConnections*, is checked functionality around using maximum allowed workers for reaching maximum performance and then is checked if after the expiration of time, when inactive threads should be destroyed, are destroyed.

In the next two tests are checked if testing of SQL statements is correctly performed. In the first case, when is passed correctly written SQL statement, is checked correct calling of an anonymous method, which should be called after a successful test. Then in the second case is tested SQL statement with fault, where is checked if was called exception handler after tests and was not called an anonymous method. Next three tests check the execution of SQL commands, where is checked both overloads of procedure *AddCommand* and whether commands from default command list are executed as well as from the worker command list.

Further, are implemented tests for checking functionality around fixing failed SQL commands in diagnostic form. In this test is added SQL command for execution with a deliberate mistake, where after its execution is first checked if this failed command appears in the diagnostic form, then is command repaired using the same buttons and edit boxes as the user usually use and finally is checked if the command was successfully executed after its correction. This test is performed with the default command list as well as with worker command list. Last two tests serve for testing execution of SQL statement with workers locking, where is checked if SQL statements are correctly executed, if anonymous methods after execution are called, exception handlers in case that exception occurs, and automatic worker unlock after the release of references to worker interface.

Figure 47 then shows how looks in Delphi if some test fail. For this test was commented code for calling the anonymous method after the execution of SQL statement and several tests caught this issue.

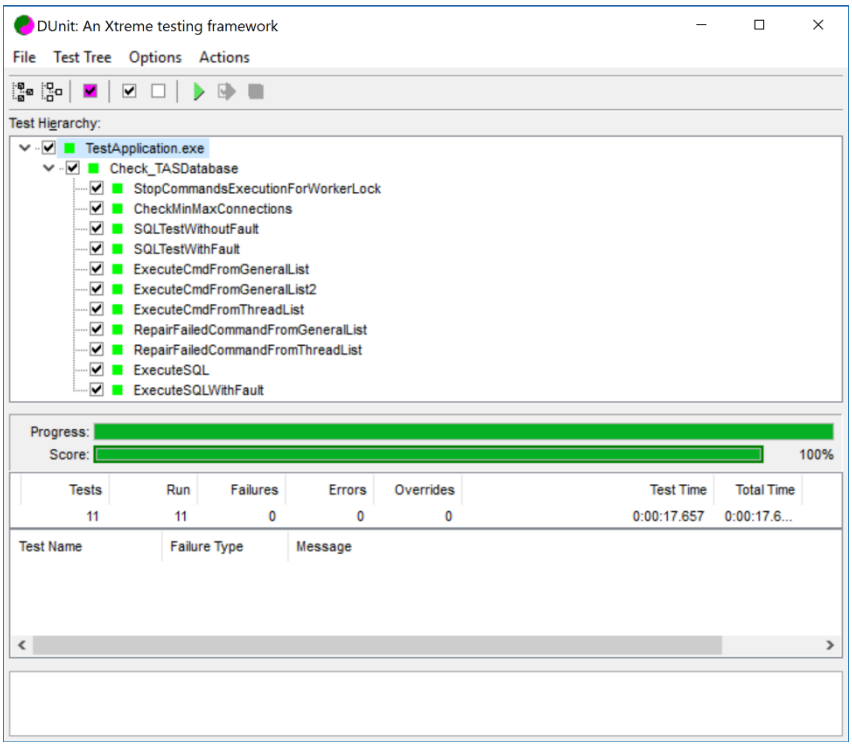

Figure 46 Unit tests after successful testing

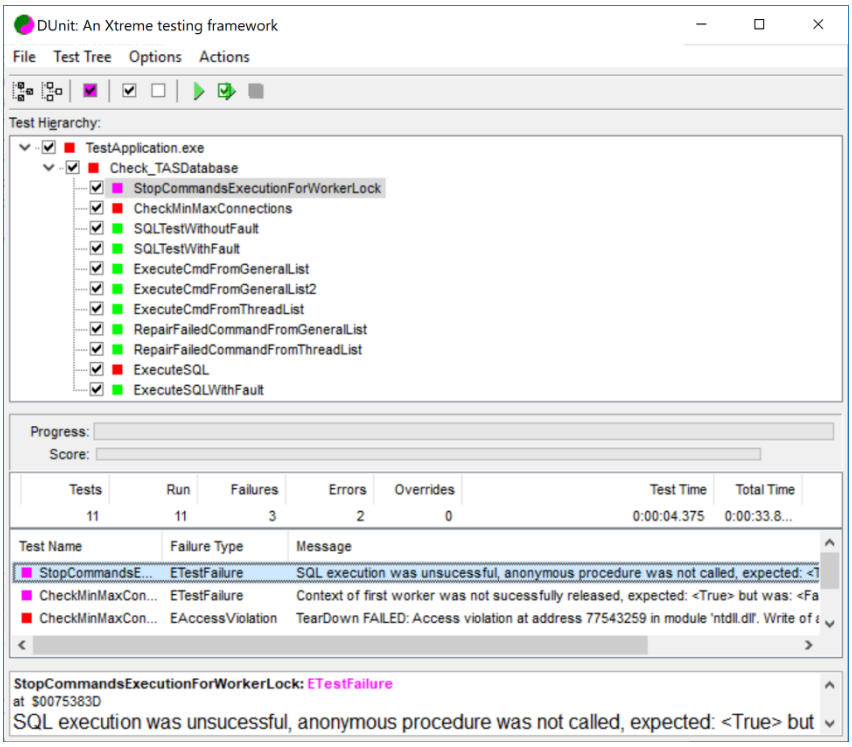

Figure 47 Unit test with issues during testing

11.2. Testing library in production

In times that was this diploma thesis written also started library testing in a production environment. The library was used in the new application which has the task to set data in PLC according to data in the database, which is refreshed each 20s. For this case is each 20s locked worker, with which is executed SQL statement which fetch required data from the database and then in the anonymous method are data from the result of execution transferred to the PLC data block, which is then sent to the PLC in communication thread.

During testing has been found one issue which was caused by the wrongly written constructor of the worker, which cause that worker sometimes tried to open database connection in a time when it has not set communication data yet. This issue was fixed and from this time was not appeared any another issue before submitting a diploma thesis.

12. Conclusion

The requested library for asynchronous SQL statements executions was successfully designed and implemented. The final solution consists of two main components which are dispatcher and workers. The dispatcher is the main component to which user accesses when he needs to execute some SQL statement. Next main job of the dispatcher is to manage workers creation, termination and execution. Workers are threads which have their connection to database and transaction which allows to them execute SQL statements in parallel form. For the user will be using of this library very comfortable. He has two options on how to execute SQL statements, where the first option is by adding SQL statement to command list and the second option is by locking workers with passing SQL statements to execution through worker interface. For the processing result of SQL statements was designed concept with anonymous methods, which are called in the main thread after workers finish its execution.

In the database library is implemented much functionality for user comfort. The first important function is silent reconnection, where the worker continuously checks if its connection to the database was not interrupted. In case that connection to the database was lost then worker immediately restores it. Next important function is automatic worker unlocking, where a feature of the interface was used. The interface is passed to the user after worker locking and he can execute SQL statements with it. The interface has own counter which counts to how many variables are assigned and when this counter reach number 0, then is worker automatically unlocked. The next worker feature is automatic destruction when it is inactive for a long time. The last great functionality for the user should be automatic SQL testing after application start. This functionality can test all SQL statements, which user register for this testing. The user by using this test can catch SQL statements which are not executed correctly, before application start. These issues could be caused for example by some modifications in the database.

For the database library was also implemented a diagnostic form that the user known the actual performance of workers, statistic information, or have the possibility to repair or delete poorly written SQL statements, which worker is not able to execute. The database library has also implemented automated tests, which automatically testing its main functionality. Now they run every day on the build server. In development sometimes happened that somebody makes some modifications, which could cause that other functionality stop works correctly. In that case, the programmer is immediately informed about it because these tests fail. In times that this diploma thesis is submitted is library also tested in production application by which was revealed one issue which is fixed and from that time was not appear any other issue.

During the implementation of the database library appeared many problems. Either topic around asynchronous execution and also topic around multithread application are not elementary on their own. Mix these two topics into one library was very difficult either from part of finding the best topology and also of the implementation part. For the implementation was firstly used prepared classes for asynchronous execution from class *TComponent*, but then was this implementation stopped. The code appeared very complicated, and so the implementation started again from zero with new topology. The second topology is that one which is described in this thesis, and it seems to

be clearer either from part of the implementation and also for the programmers who will be using this library. During the implementation of database library has also struggled with correct naming of classes, fields, or procedures so that only me would not understand how the library works, but also my colleagues and everyone who want to use this library.

The last big problem was also with finding and debugging the issues because the timing of thread is sometimes unexpected so here was appeared sporadic issues. One example is with the problem which has appeared in a production application. This function was tested several times in computer where this library was developed, but after deployment to the production server, it sometimes appeared. Just a little bit different CPU with different timing of processes switching and the problem appear. Following example could be with the deadlock in destructor, which was fixed thanks to automated tests. The tests were several times successful, but one time from approximately twenty attempts the test froze, because dispatcher has locked list with workers and waiting for their termination and meanwhile one worker waiting for the list unlock, because worker wants to notify main thread that its inactive time exceeded the allowed value and it wants to be terminated. Three days of debugging to find this sporadic issue.

Despite all mentioned problems, the database library in days of the thesis submitting works without problem in a production application and also automatic test on build server are without problems. Nowadays is also planned to start to implement this library into the essential applications of the company, which should fill up the main aim of this library which is get rid of application jamming which is caused by synchronous SQL statements execution, where is main thread blocked until is SQL statement executed. With this library, the main thread can do everything else what is necessary when SQL statements are currently executed.

References

[1] CÍSAŘ, Pavel. InterBase/FireBird: Tvorba, programování a správa databází. Brno: Computer Press, 2003. ISBN 80-7226-946-1

[2] GRAY, Jim a Andreas REUTER. Transaction processing: concepts and techniques. San Francisco: Morgan Kaufmann series in data management systems, 1993. ISBN 1-55860-190-2.

[3] BACON, Jean. Concurrent systems: operating systems, database and distributed systems. 2nd ed. Harlow: Addison-Wesley, 1998. International computer science series. ISBN 0-201-17767-6.

[4] GABRIJELCIC, Primoz. Delphi High Performance: Concurrency, Multi-threading, Memory Management, and more. Packt Publishing, 2018. ISBN 978-1-78862-545-6.

[5] KADLEC, Václav. Učíme se programovat v Delphi a jazyce Object Pascal. Praha: Computer Press, 2001. ISBN 80-7226-245-9.

[6] LEBLANC, David, Michael HOWARD. Bezpečný kód: Techniky a strategie tvorby bezpečných webových aplikací. Přeložili David KRÁSENSKÝ a Jiří FADRNÝ. Computer press, 2010. ISBN 978-80-251-2050-7.

[7] LABOON, Bill. A Friendly Introduction to Software Testing. CreateSpace Independent Publishing Platform, 2016. ISBN 978-1523477371.

[8] PATTON, Ron. Testování softwaru. Přeložil David KRÁSENSKÝ. Praha: Computer Press, 2002. Programování. Pro každého uživatele. ISBN 80-7226-636-5.

[9] KIENZLE, Jörg. Open Multithreaded Transactions: A Transaction Model for Concurrent Object-Oriented Programming [online]. LAUSANNE, 2001 [cit. 2019-04-06]. Dostupné z: http://jotm.objectweb.org/related/kienzle-thesis.pdf. ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE. Vedoucí práce Alfred Strohmeier.

[10] KIM, Sun Hwan, Mi Suk JUNG, Jun Hyun PARK a Young Chul PARK. A design and implementation of savepoints and partial rollbacks considering transaction isolation levels of SQL2 [online]. Hsinchu, Taiwan, 21 April 1999 [cit. 2019-04-06]. DOI: 10.1109/DASFAA.1999.765764. Dostupné z: https://ieeexplore.ieee.org/document/765764

[11] TONGYOO, Titiwoot, Vanvisa CHUTCHAVONG a Ornlarp SANGAROON. Object-Oriented Design Message Control Multi-Threaded Execution [online]. Busan, South Korea, 21 Oct. 2006 [cit. 2019-04-06]. DOI: 10.1109/SICE.2006.315759. Dostupné z: https://ieeexplore.ieee.org/document/4109256

[12] KIM, Ju Gyun. An algorithmic approach on deadlock detection for enhanced parallelism in multiprocessing systems [online]. Aizu-Wakamatsu, Japan, 21 March 1997 [cit. 2019-04-06]. DOI: 10.1109/AISPAS.1997.581669. Dostupné z: https://ieeexplore.ieee.org/document/581669

[13] OMAR, Faizah a Suhaimi IBRAHIM. Designing Test Coverage for Grey Box Analysis [online]. Zhangjiajie, China, 15 July 2010 [cit. 2019-04-06]. DOI: 10.1109/QSIC.2010.44. Dostupné z: https://ieeexplore.ieee.org/document/5562984

[14] LAWANNA, Adtha. The Theory of Software Testing. Intelligent Transportation Systems Journal [online]. June 2012, 35-40 [cit. 2019-04-06]. ISSN 1024-8072. Dostupné z: https://www.researchgate.net/publication/236031163_The_Theory_of_Software_Testing

[15] SAUNOIS, Lucie. Black box, grey box, white box testing: what differences?. NBS system [online]. 10 May 2016 [cit. 2019-04-06]. Dostupné z: https://www.nbs-system.com/en/blog/black-box-grey-box-white-box-testing-what-differences/

[16] FireDAC Multi-Device Data Access Library. Embarcadero Technologies, Inc. [online]. 2019 [cit. 2019-04-06]. Dostupné z: https://www.embarcadero.com/products/rad-studio/firedac

[17] System.Classes.TThread. RAD Studio API Documentation [online]. 18 August 2014 [cit. 2019-04-06]. Dostupné z: http://docwiki.embarcadero.com/Libraries/Tokyo/en/System.Classes.TThread

[18] System.Threading.TTask. RAD Studio API Documentation [online]. 16 February 2015 [cit. 2019-04-06]. Dostupné z: http://docwiki.embarcadero.com/Libraries/Tokyo/en/System.Threading.TTask

[19] Using the Asynchronous Programming Library. RAD Studio API Documentation [online]. 21 October 2015 [cit. 2019-04-06]. Dostupné z: http://docwiki.embarcadero.com/RADStudio/Tokyo/en/Using_the_Asynchronous_Programming_Library

[20] System.Classes.TComponent.BeginInvoke. RAD Studio API Documentation [online]. 4 May 2015 [cit. 2019-04-06]. Dostupné z: http://docwiki.embarcadero.com/Libraries/Tokyo/en/System.Classes.TComponent.BeginInvoke

[21] System.Classes.TThread.Queue. RAD Studio API Documentation [online]. 10 February 2014 [cit. 2019-04-06]. Dostupné z: http://docwiki.embarcadero.com/Libraries/Tokyo/en/System.Classes.TThread.Queue

[22] Asynchronous Execution (FireDAC). RAD Studio API Documentation [online]. 4 December 2015 [cit. 2019-04-06]. Dostupné z: http://docwiki.embarcadero.com/RADStudio/Tokyo/en/Asynchronous_Execution_(FireDAC)

[23] KENNEDY, John a Michael SATRAN. Synchronous and Asynchronous I/O. Microsoft Docs [online]. 05/31/2018 [cit. 2019-04-06]. Dostupné z: https://docs.microsoft.com/en-us/windows/desktop/FileIO/synchronous-and-asynchronous-i-o#synchronous-and-asynchronous-io-considerations

[24] Asynchronous I/O. In: Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 2018, 18 November 2018 [cit. 2019-04-06]. Dostupné z: https://en.wikipedia.org/wiki/Asynchronous_I/O

[25] Delphi (IDE). In: Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 2019, 6 April 2019 [cit. 2019-04-06]. Dostupné z: https://en.wikipedia.org/wiki/Delphi_(IDE)